

C++

Programming Today

Barbara Johnston

FREE CDs ENCLOSED!
Book not returnable if software
has been removed.
CDs include Microsoft® Visual C++ 6.0 Compiler,
Source code for all sample programs.

C Programming ++ Today

This page intentionally left blank

C++ Programming Today

Second Edition



Barbara Johnston
Central New Mexico Community College



Upper Saddle River, New Jersey 07458

Library of Congress Cataloging-in-Publication Data on File

Vice President and Editorial Director, ECS: *Marcia J. Horton*
Executive Editor: *Tracy Dunkelberger*
Assistant Editor: *Carole Snyder*
Editorial Assistant: *ReeAnne Davis*
Managing Editor: *Camille Trentacoste*
Production Editor: *Rose Kernan*
Director of Creative Services: *Paul Belfanti*
Creative Director: *Juan Lopez*
Cover Designer: *Kenny Beck*
Managing Editor, AV Management and Production: *Patricia Burns*
Art Editor: *Thomas Benfatto*
Director, Image Resource Center: *Melinda Reo*
Manager, Rights and Permissions: *Zina Arabia*
Manager, Visual Research: *Beth Brenzel*
Manager, Cover Visual Research and Permissions: *Karen Sanatar*
Manufacturing Manager, ESM: *Alexis Heydt-Long*
Manufacturing Buyer: *Lisa McDowell*
Executive Marketing Manager: *Robin O'Brien*
Marketing Assistant: *Mack Patterson*
Cover Photo: Alan and Sandy Carey/Photodisc/Getty Images, Inc.



© 2008 Pearson Education, Inc.
Pearson Prentice Hall
Pearson Education, Inc.
Upper Saddle River, NJ 07458

All rights reserved. No part of this book may be reproduced in any form or by any means, without permission in writing from the publisher.

Pearson Prentice Hall™ is a trademark of Pearson Education, Inc.

All other trademarks or product names are the property of their respective owners.

The author and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

ISBN: 0-13-615099-3

Pearson Education Ltd., *London*
Pearson Education Australia Pty. Ltd., *Sydney*
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd., *Hong Kong*
Pearson Education Canada, Inc., *Toronto*

Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan, *Tokyo*
Pearson Education Malaysia, Pte. Ltd.
Pearson Education, Inc., *Upper Saddle River, New Jersey*

Dedication

To Mary Jane Willis

Your courage and character inspires us to get up and walk in the morning!

This page intentionally left blank



Preface

This Book Is for You

C++ Programming Today was written for the individual who is interested in learning to program in C++. You may be

- a college student, studying for an associate's, a bachelor's or master's degree,
- a community college student, working toward an associate's degree,
- an employee interested in becoming a programmer at your company,
- an individual who wishes to learn C++ before learning Java or C#, or
- a retired individual interested in learning to write programs for fun.

Whatever your situation, if you are ready to learn computer programming, *C++ Programming Today* is for you.

How Much Math Do You Need for *C++ Programming Today*?

I do not assume that you have had any recent math courses or that you remember much of the mathematics you knew at one time. It is true that many programming applications require rigorous mathematics, but you will not need that here—just multiplication, division, addition, and subtraction. You won't find a summation using Greek letters in this book! We do cover some calculations in this text, but the necessary algebraic formulas are presented and explained.

The Examples Are Easy to Understand!

Some C++ topics can be difficult to grasp, but the examples presented here have been designed for easy understanding. We keep it simple, so that you can concentrate on the details of the C++ language. The second edition contains more than 115 complete programming examples for illustration as well, including several Practice examples at the end of each chapter. These examples help demonstrate how C++ with object programming is performed.

Light on the Math, Heavy on the C++ Concepts!

We need just a simple review of mathematics, but this does not mean that we will tread lightly on the C++ language. The book tackles some complicated programming topics including classes, objects, virtual functions, and inheritance. Pointers and references are introduced in Chapter 5 and used where needed in the rest of the book.

Great Features in *C++ Programming Today Second Edition!*

This edition includes practical, common sense programming tips. Some are listed here.

- Programmer's Do's and Don'ts; guidelines illustrating what to do and not to do in your programming endeavors.
- More than 115 complete C++ programs are spread throughout 8 chapters; they present a complete set of C++ program construction. This is a significant increase from the 1st edition. Students like to see the entire "larger" programs instead of short ones on snippets of code.
- The end of chapter material has been significantly expanded for all 8 chapters. Because the 1/e had the Lab Manual, the end of chapter material was "thin". I have really expanded all the problems so that there is a wealth of material from which to choose assignments.
- Trouble Shooting sections provide insight into common mistakes and pitfalls that programmers encounter.
- Writing code the right way is important and sometimes seeing code written incorrectly can be informative, too! Right and wrong, good and bad examples are here in *C++ Programming Today*.
- Building programs in a step-by-step manner is described.
- Tips for working with data arrays are useful, too.
- Common compiler and link errors associated with the new features are shown in each chapter. This is presented in a concise format using sample programs for the specific chapters. This feature presents the exact error that a new programmer sees, and explains what is causing it and how to fix/avoid in the future. (I did something like this in the 1/e but the 2/e is a better, well-liked format.).
- Helpful ideas concerning the type of programming error and resulting compiler warnings/errors or program behavior are given.
- A wide variety of program examples, ranging from short one-liners to multi-file, multi-object medium sized programs are included.
- Complete programs are used so that the reader can use them as starting points for further expansion.
- Four icons are used throughout the text to highlight good and bad program examples as well as troubleshooting tips.



Good Programming Practice!



Be Cautious!



Stop! Do Not Do This!



Troubleshooting Tip

C++ Programming Today Has Information You Don't Find in Other Texts

Many of the *C++ Programming Today* appendices contain information and demonstration programs for the day-to-day programmer. Summary tables for function and language usage are found throughout the text. There are several unique appendices, including:

- Getting Started with Visual C++ 2005 Express Edition, and the Visual C++ Help,
- File Input/Output showing sample programs for working with text and binary files,
- Debugging information for Microsoft Visual C++ 2005 Express Edition Debugger,
- Fundamentals for building multifile programs, and
- A keyword dictionary, containing the 63 C++ keywords, with a description and example of how the keyword is used.

The Second Edition

C++ Programming Today 2nd Edition is written with Johnston's "all the things you need, and none of the things you don't" approach to an introductory language textbook. It takes beginning C++ programming students through the fundamentals of the language and into the writing and use of their own classes and objects. Microsoft Visual C++ Express 2005 (C++ development environment) is included with the text.

What's different about the *C++ Programming Today 2nd Edition* text is that it has the reader thinking about, and using C++ classes right from the start. (The 1st edition did not broach class discussion until Chapter 9.) As this new text develops the language fundamentals, it also introduces the reader to a new C++ class in every chapter. The text shows these C++ classes in program examples and end-of-chapter assignments. Using a C++ class, making objects, calling class functions make for interesting and fun programs for the beginning C++ student. This new approach to teaching C++ enables the reader to become comfortable with classes and objects before he is asked to write one.

Classroom "Tested"

The material in this C++ text is precise, accurate, and written with a sense of humor. Reviewers and students alike comment as to Johnston's easy-to-read style. Recently, instructors from across the country have reviewed the 1st edition. These individuals had excellent suggestions, many of which have been incorporated into the 2nd edition. Also, the material in the *C++ Programming Today* 2nd Edition was test-driven by introductory C++ programming students in her courses at the Central New Mexico Community College (CNM). Beginning C++ programmers latched onto the C++ string, vector, stringstream, and queue classes, and enjoyed using them in the assigned programs. These students found the "Who's Job Is It?" and job description approach to functions and classes easy to grasp, and became proficient at looking up class methods. This tactic is incorporated in the 2nd edition of this text.

Specific Differences between First and Second Editions of This Text

The style for this book was changed so that program source code is line-numbered, comments are clearly delineated, and program output follows directly after the source code. There are many examples of larger, complete programs in each chapter. The good, bad, caution, and trouble shooting icons, margin glossary notes, and debugging information are still present.

The 1st edition had several appendices that provided important programmatic information. The 2nd edition has incorporated more of this material into the sample programs and Practice problems so that students have more examples for reference. The 2nd edition includes appendices with reference material.

A Laboratory Manual accompanied the 1st edition of *C++ Programming Today*. There will be no manual for the 2nd edition, but the number of programming problems at the end of each chapter has increased significantly. Many of these programming problems are tried and true problems that have been test-driven by Johnston's CNM students.

The 2nd edition presents function information in two chapters, and does not do a separate Pointers chapter (Chapter 4 in 1st edition). Chapter 4 in the 2nd edition covers the basics of functions, including overloading and default parameter lists. Chapter 5 presents addresses, pointers, references, and functions. This streamlined approach has been well received by students.

CD Bundled C++ Programming Today Second Edition.

The CD that comes with this text is a copy of the Microsoft Visual C++ 2005 Express Edition. This may be installed on a personal computer and used to build and run C++ programs. This software includes the Visual C++ development tools and help section as well as an excellent C++ Language Reference.

Special Note: ISO C++, not Visual C++

The programs included in this text are written to the International Standards Organization C++ standards. You may run (or modify and run) these programs on any computer system that has ISO Standard C++ software. The Microsoft Visual C++ 2005 Express software is included so that you may load it on your Microsoft Windows-based personal computer at home and practice writing C++ programs. The appendices cover how to build and debug a program in Visual C++. To create and run these programs you must use the Microsoft Visual C++ Console Application.

Life after C++ Programming Today Second Edition

Master the concepts in this text and you'll be well on your way to mastering other object-oriented languages, including Java, C#, and Ruby. You'll be ready to tackle Windows development too, as well if your programming endeavors takes you toward cross-platform development arenas such as Qt or wxWidgets.

Bon Voyage!



Acknowledgments

I wish to say thank you to my immediate and extended family and friends for their support, encouragement, and patience during the writing and production of this book. Their responses to the news of another book varied from, “Oh that’s wonderful!”, to “Oh no, not another one!” As the months passed, their questions progressed from “How is it going?”, to “I haven’t heard from you! Are you buried in your book?”, to “Is it finished yet?”, to “We don’t want to see you until its finished, then we want to see a lot of you!”

Special thanks go to individuals in the Business and Information Technology Division of Central New Mexico (CNM) Community College. Thanks to Steve Parratto and Dora Lujan, fellow programming faculty members, who patiently answered my questions and provided honest feedback concerning content, direction, and the approach for this text. The Mortgage Calculator problem is for you, Steve! The crew at the E100 computer lab deserves gold stars all around: Thank you Gary Johnson, Ben Rollag, Stephanie Chelius, Carrie Yarbrough and Donnie Frank. These individual assisted in a variety of ways, including fielding Visual C++ 2005 questions, working through C++ programs, and clarifying the final programming problems. Thanks to Carrie and Donnie for their proofreading skills and their work on the solutions to the end-of-chapter problems. Paul Quan, Associate Dean, and Dr. Lois Carlson, Dean, supported my writing by scheduling and assigning me to teach the C++ courses in which this text is targeted. Lastly, I wish to thank my students who knew they were in uncharted waters—test driving these new concepts and programs. I appreciate their patience, suggestions, and kindly worded criticisms.

I am very fortunate to have two excellent “book-kids” who dedicated their time and energy for this effort. Ms. Kelly Montoya, a CNM graduate (in computer programming, now a full-time University of New Mexico engineering student) volunteered at the start of the effort to read the chapters and work through the programming examples. Kelly managed to fit me into her schedule, cheerfully reading and re-reading the drafts. Kelly brings a no-nonsense, practical approach to the subject. She’d mark up the chapter drafts telling me to “Just say it in English!” The famous, “Tell them WHY this is important!” and “Why don’t you do this, too?” Kelly provided excellent suggestions and her understanding of what you need to learn C++ is reflected here. It was great having her knowledge and directness influence this text.

Kathleen Hayter, who is returning to CNM to pursue her programming degree, offered to read the chapters when the text at the “soon-to-be-submitted to production” stage. Kathleen undertook the arduous, short-time task of reviewing

■ Acknowledgments

the chapters for clarity and completeness, using her eagle-eye for grammar, sentence structure, and wording. She worked quickly through the chapters, smoothed the rough sentences and double-checked the figures, tables, and margin glossary entries. One evening we sat in the Frontier Restaurant, chapter drafts and laptop in hand, and double-checked all eight chapters—a task that saved me hours. I can't thank Kathleen and Kelly enough for their patience and help.

I wish to thank the computer science editorial and production team at Prentice Hall, including Tracy Dunkelburger, Executive Editor, Carole Snyder, Associate Editor, Camille Trentacoste, Managing Editor and Rose Kernan of RPK Educational Services. Tracy provided clear insight and direction for this text and I appreciated the time she spent with me. Carole was my constant guide during the development of the manuscript. She was my point of contact, a calm voice in the storm, fielding questions and helping in all aspects. Camille and Rose provided the necessary information during manuscript development so that the production phase would be a bit easier. I appreciate Rose's sense of humor when mine was waning.

There are several reviewers I wish to acknowledge. They are:

Yujiang Shan,	Southern Arkansas Univ.-Magnolia
Roseann Witkowski,	Orange County Community Coll
Cort Steinhorst,	University of Texas - Dallas
Laurie Thompson,	University of Texas - Dallas
Martha Sanchez,	University of Texas - Dallas
David Chelberg,	Ohio University Athens
John Dolan,	Ohio University Athens
Chris Lynch,	Clarkson University
Amoussou Guy-Alain,	Humboldt State U
J. Graham,	Susquehanna University
James Handlan,	Susquehanna University
James Boettler,	South Carolina State U.
Paul Wilkinson,	Pasadena City College
Steven Johnson,	Pasadena City College
Sassan Barkeshli,	Pasadena City College
Bob Boettcher,	North Harris College
Robert Lambiase,	Suffolk County CC
Melanie Sparks,	Pitt Community College
Sergio Cobo,	Broward CC - Central
Matthew Alimaghams,	Spartanburg Tech College
Dan "Dusty" Anderson,	Bluefield College
Tonya Melvin-Bryant,	Durham Technical Comm Coll
Khaled Mansour,	Washtenaw Community College
Emile Chi,	College of Staten Island
Larry Johnson,	Colorado School of Mines

Barbara Johnston



Brief Contents

Chapter 1

C++ Overview and Software Development 2

Chapter 2

Getting Started: Data Types, Variables, Operators, Arithmetic,
Simple I/O and 24

Chapter 3

Control Statements and Loops 100

Chapter 4

Functions Part I: The Basics 178

Chapter 5

Functions Part II: Variables Addresses, Pointers,
and References 244

Chapter 6

Arrays 292

Chapter 7

Classes and Objects 368

Chapter 8

Inheritance and Virtual Functions 464

Appendix A

Getting Started With Visual C++ 2005 Express Edition 527

Appendix B

C++ Keyword Dictionary 541

Appendix C

Operators in C++ 554

Appendix D

ASCII Character Codes 555

Appendix E	
Bits, Bytes, Memory, and Hexadecimal Notation	562
Appendix F	
File Input/Output	568
Appendix G	
Partial, C++ Class References	582
Appendix H	
Multiple Programs	588
Appendix I	
Microsoft Visual C++ 2005 Express Edition Debugger	600
Glossary	615
Index	625



Contents

1

C++ Overview and Software Development

2

Welcome!

- 1.1** What is C and What is C++? 6
- 1.2** What Do You Mean by *Object-Oriented*? 11
- 1.3** Structured Design versus Object-Oriented Design 13
- 1.4** Software Construction Techniques: An Overview 19
- 1.5** Troubleshooting 20

Review Questions and Problems

2

Getting Started: Data Types, Variables, Operators, Arithmetic, Simple I/O, and C++ Strings

24

The Big Picture

- 2.1** Programming Fundamentals 25
- 2.2** Terminology and Project Construction 29
- 2.3** General Format of a C++ Program 31
- 2.4** Program Data and Data Types 41
- 2.5** Variable Declaration in C++ 46
- 2.6** Operators in C++ 48
- 2.7** Miscellaneous Topics: *#define*, *const*, and *casting* 67
- 2.8** More Details on Keyboard Input and Screen Output 73
- 2.9** Get Started Using Classes and Objects, the C++ *string* 78
- 2.10** Practice! 82

Review Questions and Problems

3**Control Statements and Loops****100**

Decisions, Decisions!

- 3.1** Relational and Logical Operators 101
- 3.2** *if* Statements 103
- 3.3** *switch* Statements 118
- 3.4** Loops in General 122
- 3.5** *for* Loop 124
- 3.6** *while* Loop 128
- 3.7** *do while* Loop 132
- 3.8** Jump Statements 134
- 3.9** Troubleshooting 136
- 3.10** More Fun with C++ Classes, the *vector* Class 142
- 3.11** Summary 144
- 3.12** Practice! 144

Review Questions and Problems

4**Functions Part I: The Basics****178**

Little Picture, Big Picture

- 4.1** Functions in C++ 179
- 4.2** Functions: Basic Format 183
- 4.3** Requirements for Writing Functions 186
- 4.4** Overloaded Functions 200
- 4.5** Default Input Parameter List Functions 202
- 4.6** Local, Global, and Static Variables 204
- 4.7** More Fun with C++ Classes, the *stringstream* Class 211
- 4.8** Summary 213
- 4.9** Practice! 216

Review Questions and Problems

5**Functions Part II: Variable Addresses, Pointers, and References****244**

We're Good up to Here

- 5.1** Data Variables and Memory 245
- 5.2** Address Operator: & 249

5.3	Pointers	252
5.4	Functions, Pointers and the Indirection Operator	254
5.5	Functions and References	260
5.6	More Fun with C++ Classes: the <i>queue</i> Class	265
5.7	Summary	268
5.8	Practice!	272
Review Questions and Problems		

6 Arrays **292**

Run Faster, Jump Higher		
6.1	Using Single Data Variables	293
6.2	Array Fundamentals	294
6.3	Arrays and Functions	305
6.4	C-strings, also known as Character Arrays	312
6.5	Multidimensional Arrays	322
6.6	Multidimensional Arrays and Functions	327
6.7	Filling Arrays from Data Files	333
6.8	Summary	340
6.9	Practice!	342
Review Questions and Problems		

7 Classes and Using Objects **368**

Who's Job Is It?		
7.1	What Do We Know About Classes and Objects?	370
7.2	Writing Our Own Classes	371
7.3	Objects as Class Members	399
7.4	Class Destructors	404
7.5	Array of Objects	407
7.6	Overloaded Operators and Objects	412
7.7	Pointers, References, and Classes	419
7.8	Summary	424
7.9	Practice!	427
Review Questions and Problems		

8	Inheritance and Virtual Functions	464
	Parents and Children	
8.1	Why Is Inheritance So Important? 465	
8.2	Inheritance Basics 469	
8.3	Access Specifiers Specifics and Multiple Inheritance 481	
8.4	Inheritance, Constructors and Destructors 483	
8.5	Polymorphism and Virtual Functions 490	
8.6	Summary 500	
8.7	Practice! 500	
	Review Questions and Problems	

Appendices

A	Getting Started with Visual C++ 2005 Express Edition 527
B	C++ Keywords Dictionary 541
C	Operators in C++ 554
D	ASCII Character Codes 555
E	Bits, Bytes, Memory and Hexadecimal Notation 562
F	File Input/Output 568
G	Partial C++ Class Reference 582
H	Multifile Programs 588
I	Microsoft Visual C++ 2005 Express Edition Debugger 600
	Glossary 615

Standard C++ Library Description used in this Text

Library Name	Purpose
<algorithm>	**provides means to work on containers, including sorts and searches
<cctype>	utilities/diagnostics for single character
<cmath>	common mathematical computations including sqrt(), pow(), trig functions
<cstdlib>	provide a variety of miscellaneous utilities including random number functions
<cstring>	functions for working with character array/null terminated c-strings
<ctime>	provides functions for dealing with date and time
<fstream>	provides classes for working with file input/output
<iomanip>	provides manipulator functions that can be included in an I/O expression
<iostream>	contains classes for I/O operations, including pre-defined objects cout and cin
<queue>	*a container that holds items, items are stored in first-in-first-out format
<sstream>	stream class that works on C++ string objects
<stack>	*a container that holds items, items are stored in first-in-last-out format
<string>	provides string objects and useful functions for working with text
<vector>	*a container that holds items, it is a dynamic array

* C++ Standard Template Library Container

** C++ Standard Template Library Iterators (works on Containers)

Complete List of Standard C++ Library

<algorithm>	<bitset>	<cassert>	<cctype>	<cerrno>
<cfloat>	<ciso646>	<climits>	<clocale>	<cmath>
<complex>	<csetjmp>	<csignal>	<cstdarg>	<cstddef>
<cstdio>	<cstdlib>	<cstring>	<ctime>	<cwchar>
<cwctype>	<deque>	<exception>	<fstream>	<functional>
<iomanip>	<ios>	<iosfwd>	<iostream>	<istream>
<iterator>	<limits>	<list>	<locale>	<map>
<memory>	<numeric>	<ostream>	<queue>	<set>
<sstream>	<stack>	<stdexcept>	<streambuf>	<string>
<strstream>	<utility>	<valarray>	<vector>	



C++ Overview and Software Development

KEY TERMS AND CONCEPTS

ANSI/ISO standard
C++ class
compiler
cross-platform code libraries
functions
linker
object
object-oriented program
open source software
operators
pointers
portable language
references
software development steps
software development skill set
source code
standardized libraries
structured programming
top-down
Visual C++ 2005

CHAPTER OBJECTIVES

Introduce this text to the reader.

Present the history and an overview of the C and C++ languages.

Illustrate the concept of object-oriented programming.

Compare structured program design and object-oriented program design.

Help the programmer begin thinking about all aspects of building software.



Welcome!

You are about to embark on a wonderful journey. Learning to write computer programs with the C++ programming language will bring you a wonderful sense of accomplishment. When you become an efficient C++ developer, technical career doors will open for you and many programming paths will become available. C++ is a cornerstone in today's programming environment. Whether you are writing controllers for robots, implementing elegant user interfaces, building high-speed graphics for games, web services, interfacing into databases, or learning the C++, C, or Java programming language for Internet and Web application, C++ is where it all starts.

But beware! Programming—in any language—is not for everyone. In the journey there are often speed bumps, potholes, clear-air turbulence, and fog. A successful programmer uses many skills, including logical thinking and troubleshooting. At times, success seems to depend on just plain luck. If you are already familiar with another programming language and are now learning C++, you are aware of all these factors. If C++ is your first programming language, you will find these pearls of wisdom to be true soon enough.

Software Developer Skill Set

Aside from learning C++, there are several other skills a student must learn to be a successful software developer and to contribute fully to a development or maintenance job. A few of these skills are listed in Table 1-1.

A software developer must also be a student of other fields. For any software job, the developer must understand the underlying concepts of the software while also working closely with non-software experts on a project. For example, to write an inventory control and tracking system for an international company, the software team must have a good idea of all aspects of the business. If you are writing a program to handle bad debt for a company, you will need to know the laws and company policies concerning debt. Software developers are not expected to be experts in everything, but they should expect to keep learning new things long after school is out.

TABLE 1-1

Software Developer's Skill Set

Skill	Where It Is Used
Documentation	All software needs to be documented well and should be easy to read. Documentation is written into the source code as comments and describes what the program is doing, the logic that is used, how the user interacts with the program, and any special methods used. You should write your software assuming that it will be used for years to come. Can someone read your software and know what you were doing?
Communication	Software developers must be able to communicate with team members, customers, and end users. Software projects often go through several design stages. Formal design and requirements documents are written before any code is built.
Quick learners	Software developers often work with experts in other fields who need to have software written. The software developer must gain some level of understanding of the new concepts before he or she is able to build the software. Often programming requires the use of mathematics. The programmer should be familiar with mathematical concepts.
Debugging	It is vitally important to learn debugging skills when developing software. Debuggers aid in finding problems in software.
Troubleshooting	Debugging "broken" code is common. Good analysis/troubleshooting skills are invaluable.
Testing	Always think about how to test your software. Are you covering all the cases? Testing is one of the most important aspects of software development, but it is the most often neglected.

A Few Details Concerning This Text

Concepts, Order of Presentation Chapters 2 through 6 present the nuts and bolts of the C/C++ language including an introduction to object-oriented concepts in Chapter 1. In Chapter 2, we begin using classes provided in the C++ standard libraries and learn how to get data in and out of a program. We write short programs that use different types of data, practice writing math statements. In Chapter 3 we need to be able to have our code perform logical and conditional statements and loops. Chapter 4 teaches us how to separate our code into individual modules known as functions. We explore the various ways C++ lets us write functions, including overloading, and default parameter lists. In Chapter 5, we expand our discussion of functions by presenting an introduction to references and pointers. Pointers and references are handy little devices found everywhere in C++ code. Chapter 6 introduces the concept of an array—a list or grouping of data variables and, finally, Chapters 7 and 8 provide the tools for you to write your own classes.

Although we begin writing our own classes and using our objects in Chapter 7, we begin using several C++ classes right from the start! Chapter 1 introduces an easy way to understand classes and objects, and then we use the C++ string class in Chapter 2. In Chapter 3, we begin using the C++ vector class. These classes give us

the ability to build interesting C++ programs right away. By the time we reach Chapter 7, we are familiar with classes. The mechanics of writing object-oriented software are based on the concepts we've been studying in the first several chapters. In the latter chapters, we write several programs using many of the classes we write as well as the classes C++ gives us.

Accompanying CD with This Book One compact disc is included with this book. On it is a copy of the *Microsoft Visual C++ 2005 Express Edition*, which may be installed on a personal computer and used to build and run C++ programs. This software includes the Visual Studio C++ development tools as well as an excellent C++ Language Reference. Appendix A, "Getting Started with Visual C++ 2005 Express Edition," covers the system requirements, software installation, all of the necessary steps to create and run a simple project, and how to access the help feature. (The debugger tool is described in Appendix I, "Microsoft Visual C++ 2005 Express Edition Debugger.")

Microsoft Visual C++ 2005

Microsoft's C++ development environment

Program Source Code The source code for all the program examples in this text are available on the text's website. These examples are organized into Microsoft Visual C++ 2005 project folders, and each is located in its respective chapter folder. The project folders should be copied onto the reader's hard drive. The projects may be opened using Visual C++ 2005, or the source files can be copied and compiled in other C++ environments.

ISO C++, Not Visual C++ The programs included in this text are written according to the International Standards Organization (ISO) C++ standards. You may run (or modify and run) these programs on any computer system that has ISO standard C++ software. The Microsoft Visual C++ 2005 software is included so that you may load it on your personal computer at home and practice writing C++ programs. One appendix covers how to install and get started with this software. Another appendix covers how to use the debugger in Visual C++ 2005. We will be creating and running our programs by using the Console Application.

Program Examples, Output, Errors, and Warnings This book also includes a wide variety of program examples—short five-liners to multifile, multi-object, medium-size programs. Some are complete programs, and the reader is welcome to use them as starting points for further expansion, such as accessing the computer system time as a seed for the random number generator. You will find many examples of code that are written incorrectly and hence do not build and execute. (Sometimes seeing the wrong way is more informative than seeing the right way.) We illustrate the wrong code along with the software error and/or warning messages.

Johnston's Rules for Programmers

As a team member of a commercial firm engaged in building and developing software, I am acquainted with technology skills needed in the marketplace. And as a member of the faculty of a community college, I have learned ways to help students master these skills. Teaching beginning C and C++ students for many

years has brought an understanding of the trials and tribulations that the new C++ student encounters.

Drafts of the first edition of this text were used for four semesters by beginning and advanced C++ programming students. This second edition revision also used C++ programming students as test-drivers, who worked through new ideas, concepts, and programs. Past students also volunteered to help with this second edition. My beginning C++ students are excellent test subjects and proved to be adept at using C++ classes in their second program. They followed the introductory examples and discussions here, and were soon showing that they were apt object users.

While test-driving the material in this book, these first semester programming students were quite open about what they needed in the text to aid them in learning to program in C++. They told me what they liked and did not like, and what was helpful and what was confusing. Past students who are now working as C/C++ programmers or are continuing their education have provided even more suggestions and advice.

A common theme for all these students is their belief in the set of rules we develop at the beginning of each programming course. We refer to them as either the “Nine Rules for Programmers” or the “Johnston Rules for Programmers.” They are:

- Keep your cool (don’t get mad).
- Work when you are rested (don’t program when you are tired).
- KISS your software (keep it simple, sweetie).
- Give help/get help.
- Study and know the rules for the language (syntax).
- Learn the development environment and tools (we’ll concentrate on Visual C++).
- Understand the problem you are trying to solve.
- Build and test your software in steps.
- Save early/save often (back up your computer programming files often).

One more rule might emphasize, “Patience, patience, patience.” You can apply this maxim immediately, as several topics need to be covered before we reach the core of the C++ language. Two of my students like to say that there is no crying in baseball, but they’re not so sure about programming! It is important to follow the above rules for programming so you’re not crying at your computer. Read on, have fun! Principles that now seem disjointed will come together as you practice, practice, practice your programming.

1.1

What Is C and What Is C++?

ANSI

American National Standards Institute

ISO

International Standards Organization

The C language was developed originally by Bell Laboratories and was standardized by the **American National Standards Institute (ANSI)** in 1989. In 1990, the **International Standards Organization (ISO)** adopted the ANSI standard. You can think of the ISO Standard as the governing laws for C/C++. These ISO Standards dictate the correct form of C/C++ statements, and how aspects of the language must work.

During the mid-1990s, the C language was expanded to include the ability to build object-oriented software, and hence named C++. When a student learns the C++ language, he is learning C, too. The entire C language is wrapped up in C++. The “if” statement and “for” loop are essentially identical in C and C++. C has structures; C++ has structures and classes. Classes are necessary for objects. C++ has overloaded functions and operators, whereas C does not. Structures? Functions? Operators? At this stage it is not necessary for you to know what these things are; however, it is important to realize that C++ contains all of the C language features and more!

A Brief Overview and History of C and C++

In the early 1970s at the Bell Laboratories, C was used originally as a programming language for operating systems, including Bell Labs’ UNIX. However, C quickly became a popular, general-purpose programming language because it offered programmers the basic tools for writing many types of programs.

How C Was Named Everyone wants to know how the C language acquired its name. To appreciate how C was named, one needs to examine the historical aspects of the language. C is the result of a development process that began in the early 1960s with ALGOL 60, then Cambridge’s CPL in 1963, Martin Richard’s BCPL in 1967, and Ken Thompson’s B language in 1970 at Bell Labs. Many of C’s principal concepts are based in BCPL, and these concepts influenced the B and C languages.

The Evolution of C and C++ For many years the C software supplied with the UNIX operating system was the standard C software. Brian W. Kernighan and Dennis M. Ritchie (C’s original designers) published the book, *The C Programming Language* (Prentice Hall, 1978), which describes this version of C. In 1978, aside from the UNIX versions of C, Honeywell, IBM, and Interdata also offered application production software for the C language.

As C grew in popularity and computer hardware became more affordable, many versions of the C language were created. (Rumor has it that at one time there were twenty-four different versions of C!) Hardware vendors offered their own versions of C, but there was no standard for the language. In 1982, an American National Standards Institute committee was formed to create the ANSI Standard for the C language. This standard was adopted in 1989 and ensured that developers of C software used the same rules and procedures.

In 1994, extensions and corrections to the ISO C Standard were adopted. The new features in the C language standard included additional library support for foreign character sets, multibyte characters and wide characters. Extensions to the C language, providing an improved version of C as well as supporting object-oriented programming methodologies, were designed originally by Bjarne Stroustrup of AT&T Bell Labs. These extensions to the C language became known as the C++ language. A working draft of the ISO C++ Standard was created in 1994. Over the course of several years, many drafts of the standard were produced. People discovered ambiguities and problems in the standard. Each draft corrected previous problems and added a few new features. In November 1997, the ANSI committee published the final draft of the language—which was also approved by the ISO, making it an international standard.

C/C++ Is a Compiled Language

To build a program in C or C++, a programmer must go through many steps. Aside from defining the problem requirements and designing and entering the code on the computer, the programmer must have the computer build an executable file. We will cover all these various steps in detail later. For now, it helps to understand that when you type your program into the computer, you are entering **source code**. Source code contains the actual lines of C or C++ statements that give the program direction. A **compiler** is a computer program that reads the source code, and if the code is “grammatically correct,” the compiler produces machine code.

source code

file(s) that contain the C/C++ statements that provide program instructions

compiler

a software program that reads source code and produces object or machine code

linker

software program linking machine code and library code to form an executable file

portable language

language in which the source code need not be changed when moved from one type of computer to another

cross-platform code libraries

C++ libraries that contain classes for writing programs for many types of computers

wxWidgets

an open source C++ framework for writing cross-platform source code

C++ provides libraries containing classes and functions that a programmer can use in his programs. For example, an input and output library provides tools for writing messages to the screen and receiving values that are entered from the keyboard. The next build step uses a **linker**, which literally links, or hooks, the machine code and library code together and binds it into an executable file—one that “runs” the program. Figure 1-1 illustrates these steps.

If you write a program in ISO C++ for a personal computer, it must be compiled and linked on a PC. The executable file issues commands directly to the processor when the program runs. If the program is to run on a Sun Microsystems Computer with a SPARC processor, the code must also be compiled and linked on that type of machine. Note: the ISO source code will not need to be changed. This action of requiring only ISO C++ source code to be compiled and linked on various types of computers is known as code portability. A standardized language is a **portable language**, meaning that the source does not need to be changed when moved from one type of machine to another. Figure 1-2 illustrates how the source code in Figure 1-1 needs to be re-compiled and re-linked if it is to execute on two different types of machines.

Two additional points must be noted. First, if a programmer uses custom libraries (in her code) that are specific to a certain operating system or machine, this code will not be portable across machines. For example, Visual C++ 2005 provides customized libraries for developing Microsoft Windows-specific programs; likewise, Apple has libraries for building programs for the Mac. If the program source code uses these operating system-dependent classes and functions, the code will not be portable to other types of machines.

Second, new cross-platform libraries are being developed that allow programmers to write cross-platform code. **Cross-platform code libraries** allow the programmer to write code that can then be compiled and linked on various types of machines. In the “old days” if a programmer built a program using Microsoft Foundation Class libraries, that program would compile and run only on a Windows PC. Now a programmer can write a program using the cross-platform libraries and that program can compile and run on different types of machine operating systems, including Linux, MacOS, and Solaris C. This programming paradigm is a great leap for software developers who wish to build programs that run on several types of machines. **wxWidgets** is an open source C++ Graphical User Interface (GUI) framework for building cross-platform C++ programs. The website www.wxWidgets.org provides information, downloads, and resources. Another cross-platform environment is Qt. Its Website is www.trolltech.com.

```

// File : AddTwoNums.cpp
// Program that adds two numbers.

#include <iostream>
using namespace std;

int main()
{
    int A, B, C;
    char Enter;

    cout << "\n Please enter 2 numbers";
    cin >> A >> B;
    C = A + B;
    cout << "\n The sum is " << C;
    cout << "\n Hit an Enter key to exit.";
    cin.get(Enter);

    return 0;
}

```

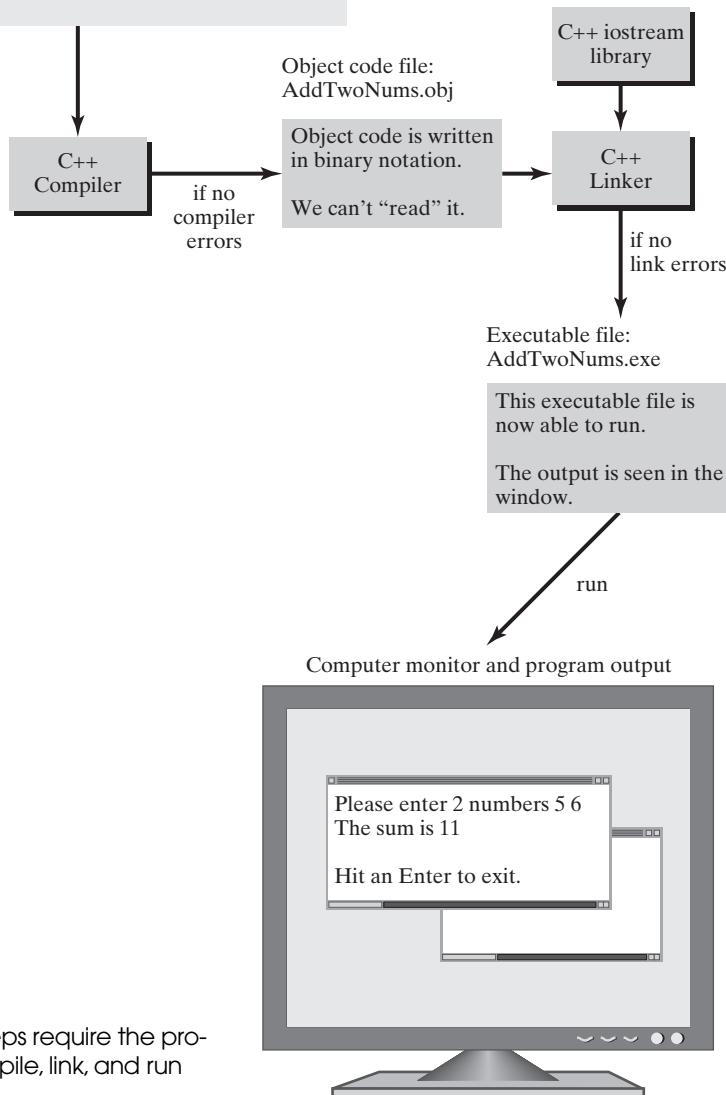


Figure 1-1

Program build steps require the programmer to compile, link, and run the program.

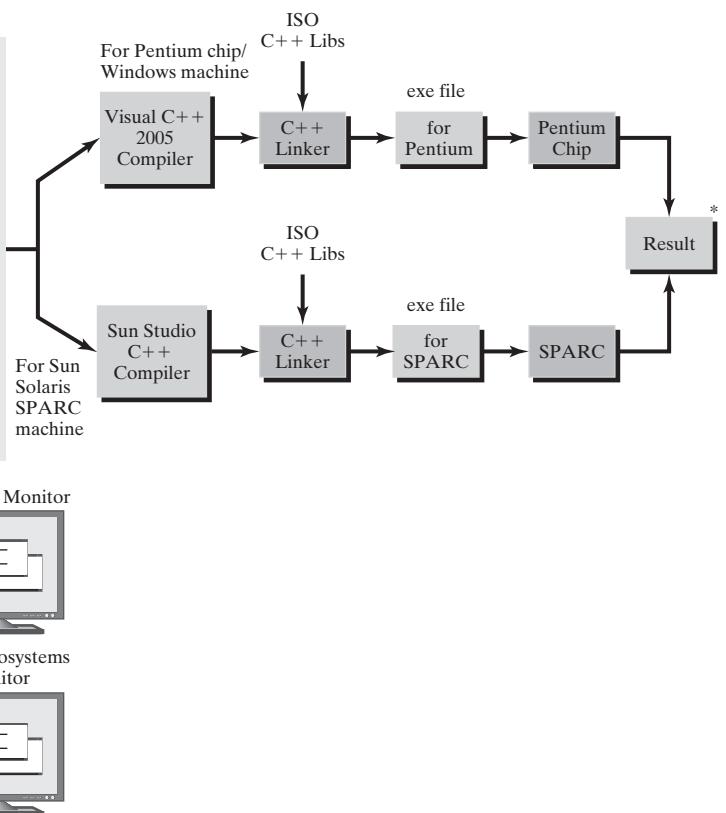
```
// File : AddTwoNums.cpp
// Program that adds two numbers.

#include <iostream>
using namespace std;

int main()
{
    int A, B, C;
    char Enter;

    cout << "\n Please enter 2 numbers";
    cin >> A >> B;
    C = A + B;
    cout << "\n The sum is " << C;
    cout << "\n Hit an Enter key to exit.";
    cin.get(Enter);

    return 0;
}
```

**Figure 1-2**

A C++ program must be compiled and linked on its target machine because the C++ executable file “talks” to the processor directly.

Open Source C++ libraries

source code that is open and free for programmers to take, use, and modify

pointers

variables that “point” to other variables

references

variables that “refer” to other variables

operators

symbols that direct certain operations to be performed, such as + for addition, = for assignment

Open Source C++ libraries provides the programmer with a wealth of C++ software. Open source software is free of charge and available to C++ programmers who can then use and/or modify the code. You can download the libraries and compile them on your machine. Working with open source allows the programmer to develop software as a community, all following licensing rules and similar methods of software construction. Members of the open source community share knowledge and code, and the information is open to everyone. As you progress in your C++ education, you may use and contribute to open source projects.

Why Do Programmers Love C++?

The C++ language provides the programmer with a wide variety of useful and powerful tools. These tools include many different types of data “containers” for handling program data—including numeric (both whole numbers and numbers with decimal precision) and character (text) data. C++ lets us use **pointer** and **references** that are variables that “point” or “refer” to other variables. There are a large number of **operators** (symbols that perform certain actions) in C++. For example, the + symbol adds numbers, and the * multiplies numbers. Basic decision

and control structures enable a program to branch to different code statements or loop through statements.

In addition, C++ allows programmers to write their code in discrete modules, known as **functions**. A function is a chunk of code that performs a task for the program. These functions enable a programmer to organize his code and separate tasks logically. An entire group of functions can be grouped into separate files or even built into libraries, enabling procedures to be organized logically. For example, if your program contains many mathematical calculations, it is possible to separate each calculation in its own function and place all the functions in a separate “Calc” library. This action helps to keep your program size from becoming too large and unwieldy.

C++ also provides the programmer with the ability to customize his own data variables so that the program data can be organized and grouped as a single unit. Don’t forget that there are many **standard libraries** that provide useful functions for mathematics, file input/output, text operations, and other tools. The C++ language also provides the **Standard Template Library** that contains software utilities and classes that are built and ready for use by the C++ programmer.

The best part of the C++ language is the fact that it provides us many classes, and that we can write our own classes as well! Once you get the “hang” of object-oriented programs, you will never want to use any other programming technique. What’s more, we can use classes to create new classes. It is possible to use a class as a “parent” or base class and create a “child” or derived class. The technical term for this is inheritance. If C++ is your first programming language, the information in this section may be a bit confusing at first. If you are here to learn C++ with other language experience, we hope that this short overview helped you gain a feeling for the tools and features found in the language.

functions

discrete modules or units of code that perform specific tasks

standard libraries

libraries included as part of the C++ language

Standard Template Library

libraries that include blueprints of classes that are ready to use

1.2

What Do You Mean by Object-Oriented?

Simply put, **object-oriented programming** means that the program is based on real-world things (objects)—designed and built to model how these “real” things interact—instead of having the program design based on how the data flows through a program. If you are an experienced structured programmer, you may find this object business a new and very different way of thinking about program design. If you are new to the field of computer programming, you may find thinking in terms of objects a natural way to lay out program components.

object-oriented programming

program design based on the use of real-world items (objects) and the manner in which these objects interact

A C++ Program Is Not Automatically Object-Oriented

If you write a program in the C++ language, it may or may not be an object-oriented program. *For a program to be object-oriented, it must contain objects.* In the old days (before object-oriented software was available), C programs were designed according to data flow and what had to happen to the data. A common design scenario is this: start the program, find and open the data file, read in the data from the file, make use of the data, calculate the necessary output, write a report, and close the data files. This type of program design is known as **top-down** or **structured**. Of course, it is possible to write a structured program in C++, but it is not object-oriented.

structured or top-down programming

program design based on the flow of the data through the program

An Easy Example of an Object-Oriented Program

An object-oriented program design takes a completely different approach from the top-down one. Instead of the step-by-step data flow through the program, an object-oriented design requires that the programmer identify the items (often real-world things) that are needed to perform certain tasks in the program. These items (objects) perform the various tasks for the program. It may be helpful to visualize different people as objects who perform different jobs. Each person (object) has a certain job description plus the necessary tools to perform his tasks. This job description is contained in what is known as a **C++ class**. An instance of a class is an **object**.

C++ class

fundamental unit in object-oriented programming that contains the “job description” for a program object

object

single instance of a class

For example, suppose we need a program where we need to read data from a file and perform some sort of math on the data and write the answers to a different file. This programming scenario can be built with two “worker-objects.” First, we create the “job descriptions” for our objects. This task is accomplished by using class descriptions. (Chapter 7 is where we learn how to write our own classes.) Our ReaderWriter class has the information, tools, and know-how for opening, reading from, writing to, and closing data files. Our second “worker” is a Calculator. The tasks of a Calculator class are to perform whatever mathematics is needed on the data and produces “answers.” Let’s have a ReaderWriter object named Martha, and a Calculator object named Don.

As our program runs, Martha opens and reads data from the files. She passes the data to Don. Martha doesn’t care about anything else in the program. Files are her whole life. Don, on the other hand, doesn’t care about files; he simply waits for the data to be passed to him. He performs the math and determines the “answers.” Once the answers are passed back to Martha, she writes the answers to a new file and closes it. Martha worries about file input and output; Don worries about calculations.

Object-Oriented Software Is Better

The C++ language is the C language with the additional necessary features and extensions the programmer needs to build object-oriented software. Object-oriented design and implementation is a more natural way to think about problem solving. Using objects, a team of programmers can identify and assign program tasks to different classes. The program then creates the required number of objects. Each object has its job description contained in a class definition. It is relatively easy to modify the class and thus correct or change an object’s task. Object-oriented software techniques have grown in popularity because this process revolves around a well-defined approach and lends itself to producing maintainable, reliable, and reusable software.

Commercial software vendors have a wealth of software tools that C++ software developers can purchase and incorporate into their own projects. Graphics packages, file translators, and complicated computer-aided design tools are just a few. What is so wonderful is that these commercial software products are all object-oriented based. Let’s say you need to read a JPEG image file into your program. You can buy a library that contains the classes needed for reading and writing JPEG files. After you buy and install the C++ libraries on your system, you simply include the library in your file, make a JPEG reader object in your code

(we can call it Bud), and Bud brings with him all the translator tools we need. We simply ask Bud to perform the reading tasks. (Of course, we need to know how to make Bud work correctly, but that will be documented with the software.) We save a lot of time and money by using pre-built classes in our programs. Isn't that great?

Software development projects have a notorious history of being delivered late, exceeding the budget, and not meeting the desired specifications. Then, once the software is in place, if the people who wrote it are not available to maintain and modify it, it is nearly impossible to correct errors or to add new features. Correctly designed and implemented object-oriented code makes software easier to maintain, modify, and expand.

Programmers need to understand that just using the C++ language does not ensure perfect code. It is easy to write “spaghetti code” in C++. (*Spaghetti code* is a term used by programmers for code that is impossible to follow and understand—like trying to unravel a single strand from a plate of spaghetti.) For programmers to learn how to write object-oriented software correctly, the software must be carefully designed and use object-oriented principles. Later chapters in this text will discuss in more detail how to design object-oriented programs.

1.3

Structured Design versus Object-Oriented Design

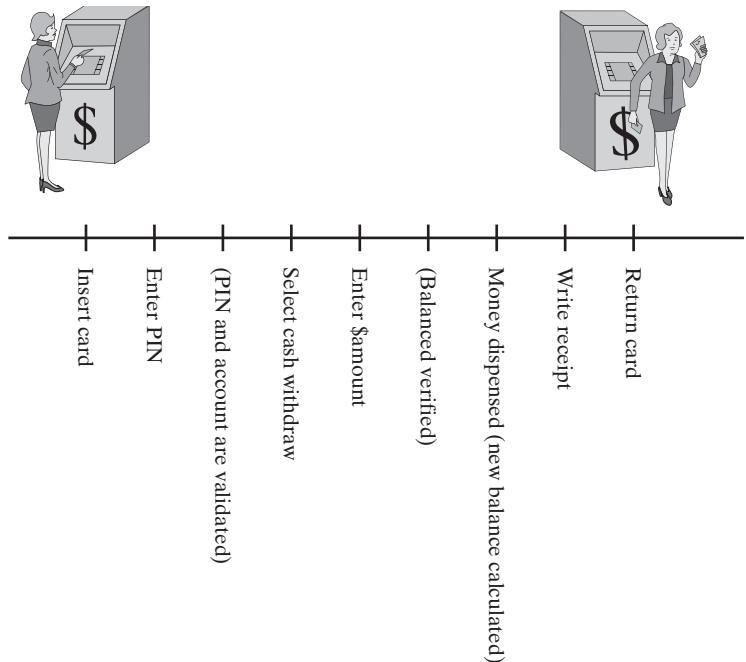
Let's do two examples in both a structured format and an object-oriented format. These examples do not contain C++ code. We just lay out the design and flow of the program. Need cash? Where is the nearest automatic teller machine (ATM)? Here we model a few of the ATM tasks.

ATM—Structured Approach

To design an ATM program in a structured, top-down approach, we need to list the steps a customer takes to use the ATM. Once the steps are identified, we determine what routines are needed. Figure 1-3 illustrates a “timeline” of activities that occur in order for our bank customer to obtain cash from the ATM. In this example, we are not showing all of the logic, such as if the PIN is invalid, or there isn't enough money in the customer's account. If we were to write a program that follows this sequence, we'd concentrate on the customer steps, and not the different components performing these tasks.

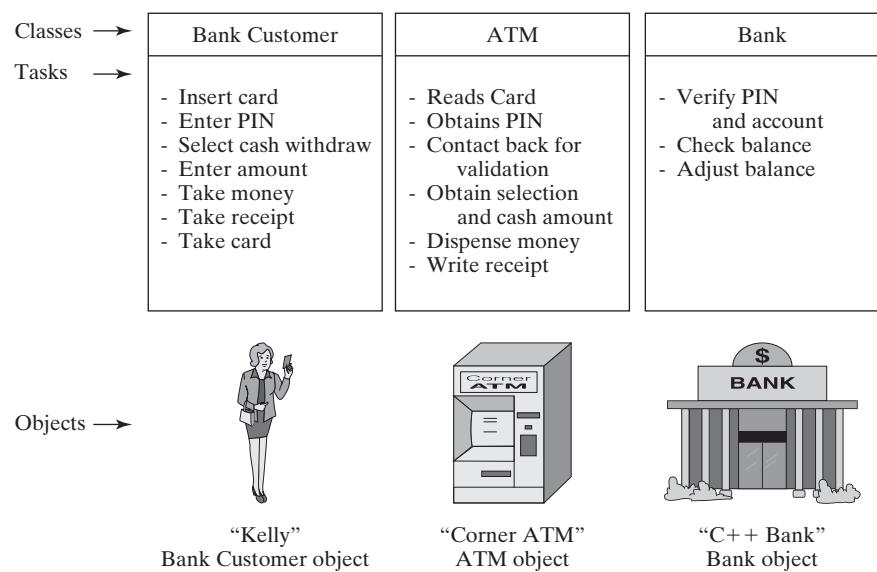
ATM Object-Oriented Approach—Who's Job Is It?

An object is a real-world item that has certain characteristics and behaviors. In our ATM example, we have three different real-world items, a bank customer, an ATM, and the bank where the customer's account is located. Instead of concentrating on steps the bank customer takes to obtain her cash, we examine the ATM program in terms of the tasks that the customer, the ATM, and the bank perform. The end result is the same, i.e., the customer obtains her cash. However, we'll look at how this act of obtaining money is an interaction between these three objects.

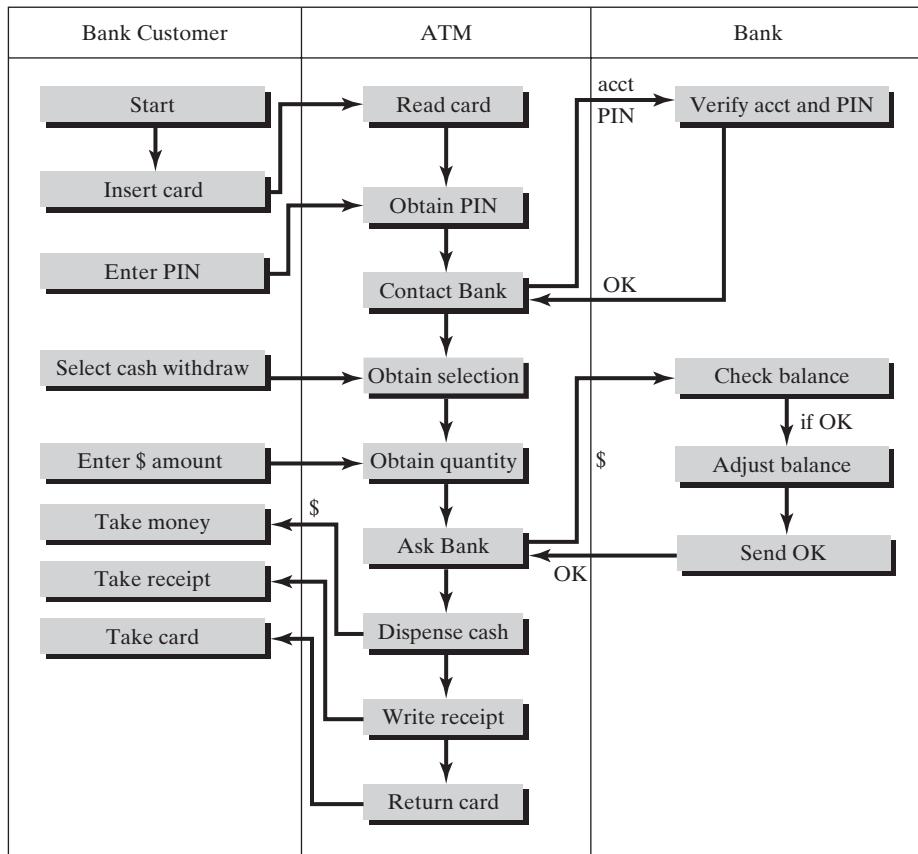
**Figure 1-3**

A structured approach to withdrawing money from an ATM. This “timeline” shows how we concentrate on the steps the customer takes to obtain her cash.

In order to design our ATM program with objects we need to ask ourselves, “Who’s job is it?” Look at the ATM timeline in Figure 1-3, and ask: “Who’s job is it to insert the card?” To read the card? Verify that the account and PIN numbers are accurate? Dispense the money? Keep track of the customer’s bank balance? It becomes obvious that the various tasks are easily separated. Figure 1-4 shows our three individuals (or objects) and the jobs they perform.

**Figure 1-4**

An object-oriented approach to withdrawing money from an ATM requires identifying the different individuals and the jobs they perform.

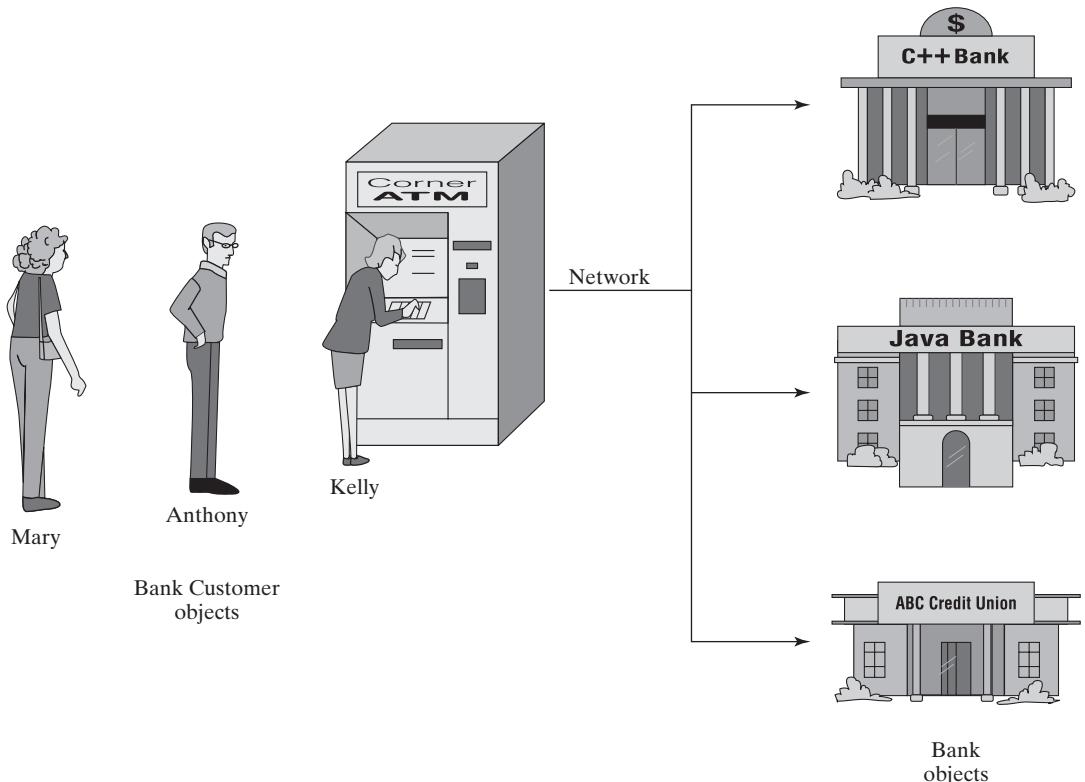
**Figure 1-5**

In an object-oriented program, the objects interact to obtain the desired results. Follow the arrows and see that our bank customer obtains her money.

As you look at Figure 1-4, there are three boxes, one for the Bank Customer, one for the ATM, and one for the Bank. Listed inside each box are the tasks (or job descriptions) that the individual performs in this money-getting activity. By writing the individual job descriptions like this, we are laying out the class. If Kelly has her account at the “C++ Bank,” Kelly is our Bank Customer object and she performs those tasks listed in the Bank Customer box. The Corner ATM is the ATM object. It performs the ATM’s jobs.

Once the individual job descriptions (classes) and their tasks are identified, we can see how they interact in order to accomplish the desired result. The steps in the timeline in Figure 1-3 are still done, but in an object-oriented approach, the objects perform the various tasks. Figure 1-5 shows the program flow for Kelly, our Bank Customer, obtaining her cash from the Corner ATM. Data items are passed between the customer and the ATM, and the ATM and the bank.

By designing a program with these individual code units, it makes it easy for us to apply this to all bank customers, ATMs, and banks in general. Figure 1-6 illustrates how two Bank Customer objects are waiting for Kelly to complete her transaction at the Corner ATM. Each person waiting in line will perform the same tasks (following the same job description) as Kelly. Perhaps these individuals have their accounts at different banks. We don’t need to write different job descriptions for the different banks, as they all perform the same tasks.

**Figure 1-6**

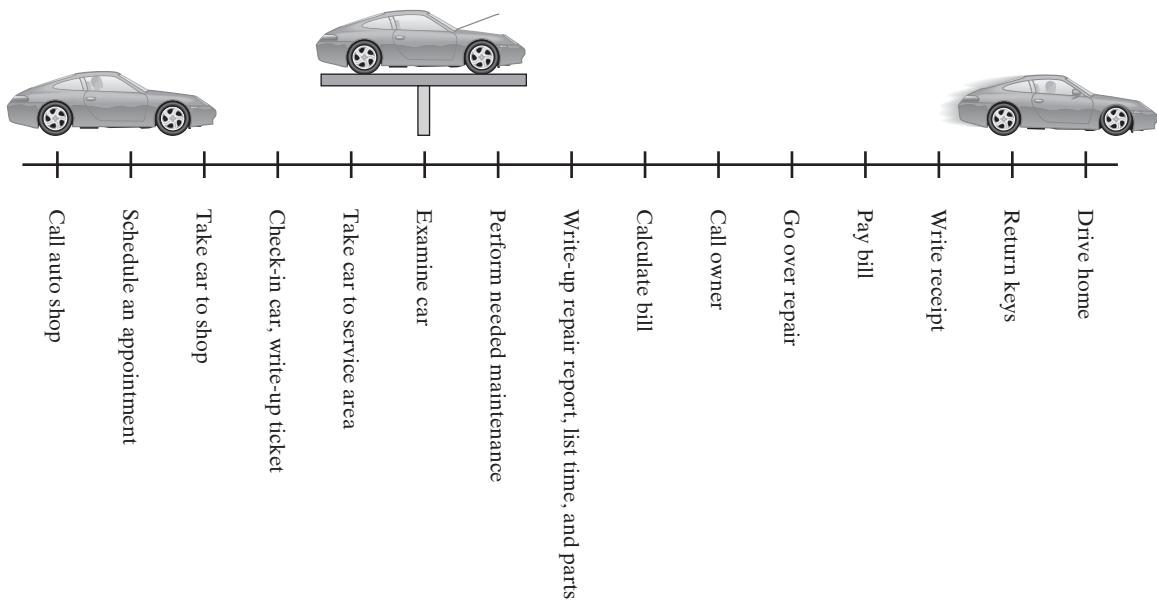
The Bank Customer objects perform the tasks defined in the Bank Customer class.

Car Maintenance—Structured Approach

Let's look at one more example that compares how programmers might design a solution using a structured approach and then using an object-oriented approach. In these two programs, we model the activities involved in having your car serviced by an automobile repair and service shop. In a structured approach to software, the central focus is on the car itself, and what must be done to achieve the desired goal. In this case, the data is the car. Figure 1-7 shows the “timeline” of steps required for servicing the car. Notice how we concentrate on *what* is happening to the car, not *who* is performing tasks concerning the car.

Car Maintenance with an Object-Oriented Approach—Who's Job Is It?

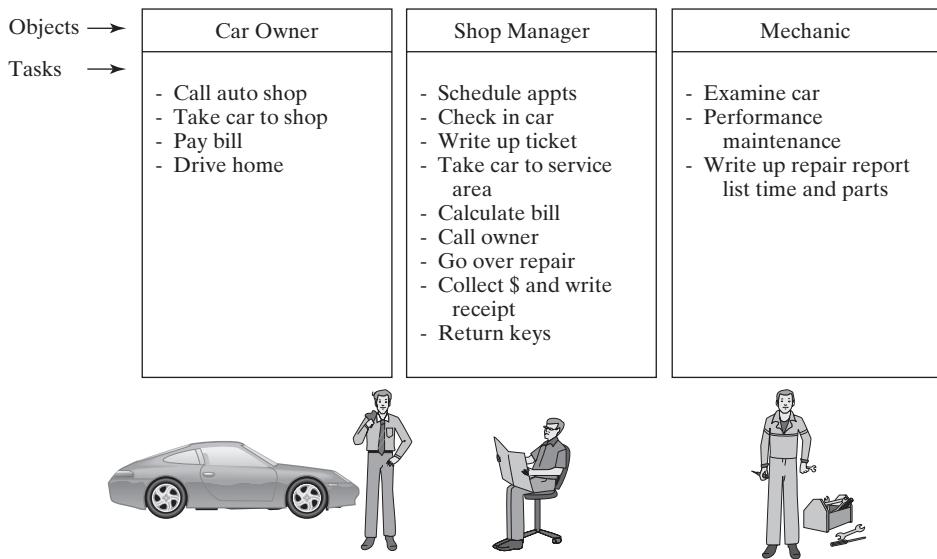
Who are the three individuals involved in this automobile activity? The owner? Yes. The mechanic? Yes. In this example let's also have an auto shop manager. Remember, in an object-oriented program you identify the individuals and the tasks that each performs to achieve your programmatic goals. Here we are able to separate the car maintenance tasks into three classes, Car Owner, Shop Manager,

**Figure 1-7**

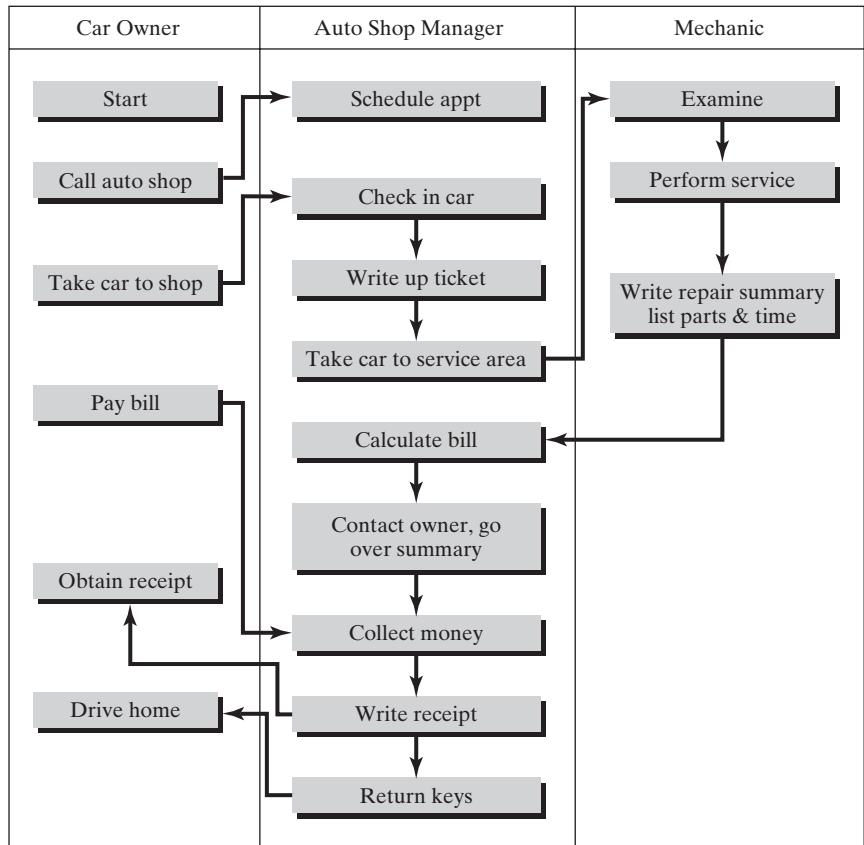
A step-by-step design for a car maintenance program. In this structured approach, we focus on the activities associated with the car.

and Mechanic. Figure 1-8 shows our three individuals and their “job descriptions.” In this figure, we see that Bob owns the car—he is the Car Owner object. Eric is the Shop Manager object, and Todd is the Mechanic object.

Our Car Owner object interacts with the Shop Manager object for scheduling an appointment and then checking in the car on the day of service. Once the car is

**Figure 1-8**

Three individuals and their “job descriptions” for car maintenance tasks.



checked in, the manager delivers the car to the service area where the Mechanic object then does the routine maintenance tasks. Upon completion, the mechanic fills out the service report and the manager then calculates the bill. The manager explains the bill to the owner when he arrives to pick up the car. After the owner pays the bill and receives the receipt and the keys from the manager, he leaves with his newly serviced car. This logical sequence of events is diagrammed in Figure 1-9.

If you compare the timeline in Figure 1-7 with the object version in Figure 1-9, the timeline seems more straightforward than the circles and arrows snaking across the page. But in the object version the objects' duties are well defined, and a programmer could concentrate on just building any one of the individual classes without worrying about the other two. Also, the tasks for the auto shop manager are very similar to an office manager for a doctor's office. The unit of code written for the auto shop manager could be used in another program with minor modifications. This illustrates the beauty of writing programs using classes—"job descriptions" for the various components of your program.

We will learn how to write these "job descriptions" and how to make the objects interact in our programs. When you are writing a "job description" in C++, you are actually writing a class. An individual in a program that is performing the job described in the class is an object. An object is an instance of a class. Before we begin writing

classes and using these objects we will practice using C++ classes provided to us in the standard libraries. In fact, we'll all be old pros at using objects when the time comes for us to write our own classes!

1.4

Software Construction Techniques: An Overview

Imagine that you are having your dream house built, and you have hired a contractor. The first day on the job the contractor shows up at your vacant lot and drops off piles of lumber, bags of cement, and rolls of wires. He leans the roll of your new living room carpeting against a tree. He stacks the windows on top of each other in the back, has the sheet-rock, electrical fixtures, plumbing items, and doorknobs lying on the dirt. Then he asks you where you want the kitchen. Moreover, he brings his crew to the site of the new house, and they are asking each other how to use the tools, reading how-to books, and looking at the wires and electrical outlets, wondering how they hook together.

Sounds silly? This scenario should never happen! The contractor has a blueprint of your house, and he knows what you want. He knows how much lumber and cement and wire to have on hand. He doesn't have items delivered until it is the right time to use or install them. His builders know how to use equipment properly and understand construction techniques. The contractor knows there is a certain order in which things must happen. He makes sure the cement is dry before putting up the frame!

Building software and building houses take similar approaches. Before you start hammering two-by-fours together on your house frame, or writing your functions, you must know what you want to build. What are the requirements and what is the program supposed to do? Just as the contractor builds your house in logical steps, software must be built in logical steps, too.

As students first start learning to program, they often make common mistakes with disastrous results. A student who would never build a house without a blueprint may sit down at the computer and start building software without a plan. Just as a contractor never will put on the roof until he is sure the walls are strong enough to hold it, students never should hook together complicated software modules without testing each piece.

How Not to Program

It is important to learn how to crawl before walking, and to walk before running. Much of this text covers the C++ fundamentals of crawling and walking. In watching babies learn to crawl, walk, and run, adults sometimes have to let them fall a few times. There are all sorts of lessons awaiting beginning C++ programmers. However, just as a responsible parent will keep a watchful eye on the toddler, it is only right to provide beginning C++ programmers with a few pointers.

The following chapters of this book provide troubleshooting information concerning the language. Table 1-2 describes many steps to avoid while programming. I have compiled this table after watching many, many beginning C++ programmers tackle programming assignments in the wrong way.

TABLE 1.2

Don'ts and Do's for Programmers

Don't	Do
Don't go directly to the computer, start typing in code, and assume you will be able to figure out the program as you go along.	Do write down on paper what you need to do.
Don't avoid testing the easy stuff.	Do test simple functions even though they may seem obviously correct. Spending five minutes checking a two-line function may save you hours of work later.
Don't depend on the compiler to ensure that your code is written correctly. The compiler will allow code to compile but will result in run-time errors.	Do understand every single line of code you enter into your program.
Don't avoid comments in your code. You will be amazed how something that seems so clear is not clear a few days later.	Do write simple, clear comments explaining your program logic.
Don't type in your entire program before compiling. Often one or two errors will result in many compiler errors.	Do build your program in steps, stopping and testing each step as you go.
Don't type in random braces if your program won't compile.	Do indent your code and line up your braces.
Don't get mad! When you find yourself getting frustrated with the program, it's time to leave it alone for a while. Go get a soft drink or coffee, take a walk, do the dishes.	Do work on your program when you are rested and it is quiet. Programming requires concentration, and interruptions often cause you to lose your train of thought.
Don't wait to start your program until the last moment, even if you do work well under pressure.	Do plan to work on your program over the course of several days. This strategy will give you time to rethink problems and refine the work as needed.

1.5

Troubleshooting

If your car does not start, what do you do? You follow a logical set of steps to determine the problem. Does the car try to start or does it just make a clicking sound? Is your battery dead? Do you have enough fuel? Is there a loose wire? Tracking down a software problem also involves asking simple questions and examining the program to find the error. You must understand the basic types of errors—which we'll spend a lot of time discussing in this text. If you have an idea what type of error you're getting, you'll be on your way to fixing it.

Claire's Number One Rule

A former student and now senior programmer at a large corporation gave me this piece of advice to add into the book. She said her #1 rule when she is working on software is that if you make a change to your code, test that change! No matter how

simple the change might be, test it! Sometimes changes introduce new problems. If you've made several changes in your code without testing them, you may end up with new problems! You then have to backtrack and undo the changes to find the problem.

The computer is unforgiving at times and does exactly what you tell it. Many mistakes can be avoided if a student learns and follows the programming rules and try not to fall into the habit of doing the "don'ts" listed in Table 1-2.

REVIEW QUESTIONS AND PROBLEMS

Short Answer

- 1.** When and where was the C language invented?
- 2.** Why was it important for the ANSI committee to standardize the C language?
- 3.** What type of programming techniques does C++ provide that the C language does not?
- 4.** For what purpose was the C language originally designed?
- 5.** What is meant when you are told to build and test your software in steps?
- 6.** If a feature is found in the C language, is that feature in C++? Is the reverse true?
- 7.** Why is it a bad idea to start a software project by first sitting down at the computer and entering source code?
- 8.** Why is it often heard that the UNIX operating system looks a lot like C?
- 9.** Why is it important to learn all the tools in your development environment?
- 10.** Name three skills programmers must have besides the ability to write software.
- 11.** Why do you have to recompile your source code if you move it from a Microsoft Windows machine to a Mac running Mac OS?
- 12.** Search the web for "Graphics Libraries in C++." Find three products that provide C++ classes that you can incorporate into your C++ program. Give a brief description of the tools these libraries provide.
- 13.** Search the web for "Image Processing Libraries in C++." Find three products that provide C++ classes for your programs. List three file formats that each library support. Indicate if they are Open Source or available for a fee.
- 14.** Search the web for "Open Source C++." List three open source implementations and state their purpose.
- 15.** What does the acronym ALGOL represent? When was the language developed? It was the first language to do what? (List three firsts.)
- 16.** The B language was implemented for what specific machines? Was it in use after the C language was standardized?

17. If you write a C++ program using the ISO C++ libraries that runs on a Windows PC, is it possible to move your program over to a computer running Mac OS? Explain why and how you would accomplish this.
18. If you write a C++ programming using operating system dependent libraries, will you be able to re-compile and re-link it on a different operating system? Explain why.
19. Some people jokingly ask, “Is C++ really a B-?” Use the web to look up information on the B language and explain your answer to this question.
20. The ISO committee began to standardize the C++ language in 1994, but didn’t deliver the “final” standard until November, 1997. Why do you think that it took several years to complete this task?

Problems

Questions 21–25 describe a situation where a service is rendered to an individual. The individuals performing the services are described. Using the ATM and Car Maintenance examples in this chapter as a guide, draw the “boxes” showing the “job descriptions” and then draw a diagram illustrating how the objects interact. Refer to Figures 1-4, 1-5, 1-7, and 1-8.

21. You are going to the doctor for a check-up. You need to call to make an appointment. When you arrive, you check in at the desk. The nurse calls you back and takes you to the examining room. She takes your blood pressure and other vital signs. The doctor comes in and performs the check up. After he is finished, you get dressed and then check out with the receptionist at the front desk.
22. You take a package to the post office and choose to send it first class with delivery confirmation. The postal clerk calculates the rate, stamps the package, and tosses it into the bin. A postal worker takes the package and puts it into the appropriate mailbag and routes it off to its destination. Once at the destination, a postal worker sorts the package into the correct postal carrier’s pile, and it is delivered to its destination. Assume that the clerk is a separate job from the postal worker (i.e., the worker inside the post office who handles organizing the in-coming and out-going mail.)
23. You stop at your favorite fast food restaurant and place your order for lunch. There is a person who takes your order, your money, and your name. The cooks read the monitor in the kitchen and prepare your food. Another individual gets your drink, and tray of food from the kitchen, calling your name when your order is ready.
24. Your dog is due for her rabies shot and heartworm test. You call your vet and make an appointment. When you and your dog show up for your appointment, you check in with the receptionist. The vet comes out and calls your name when it is your turn. She examines your dog, and gives your dog the rabies shot. She then takes the dog into the back so that the technician can weigh your dog and draw blood for the heartworm test. The technician examines the

blood and runs the other heartworm tests. The tech brings your dog back to you. The vet prescribes the correct heartworm preventative. You pay the receptionist before you leave the office.

- 25.** You need to buy 12 bags of mulch for your greenhouse and garden, so you go to your local home and garden center. Once inside, you ask the person at the information counter the location of the mulch. You find a flat cart and the mulch. Next, you find a strong young man who works at the center to help you load your 12–50 pound bags onto a cart and have him push it to the checkout stand. You pay the cashier for your mulch and the young man wheels the cart out to your truck. He loads the bags into the back of the truck. He then takes the cart back into the store and you drive home.



Getting Started: Data Types, Variables, Operators, Arithmetic, Simple I/O, and C++ Strings

KEY TERMS AND CONCEPTS

algorithm
associativity
bit and byte
case sensitive
comment
data cast
data type
`#define`
function header line
identifier naming rules
`#include`
keywords
`lvalue` and `rvalue`
operators
precedence of operations
preprocessor directive
string class
syntax
values
variable declaration
whitespace

KEYWORDS AND OPERATORS

`char, const, double, float`
`int, long, short, unsigned`
`* / % + - =`
`+= -= *= /= ++`

CHAPTER OBJECTIVES

- Indicate how to begin designing software by developing a step-by-step approach (algorithm) to a solution.
- Introduce new terminology and software construction fundamentals.
- Present a general format for a C++ program.
- Show how to write comments in C++ source code.
- Understand the concept of a data type in the C++ language.
- Demonstrate how to declare and to use a variable in a C++ program.
- Explore the various C++ operators and how these operators are used in a program.
- Illustrate the correct and incorrect methods of coding arithmetic in C++.
- Notice how a C++ program writes data to the screen and receives data from the keyboard.
- Describe compiler errors and warning messages.
- Present the C++ string class.
- Introduce the four icons to the reader:
- Good Programming Practice!
- Be Cautious!
- Stop! Do Not Do This!
- Troubleshooting Tip





The Big Picture

Programs and programming languages take many forms. There are programs that operate on mainframe computers maintaining airline reservations for hundreds of flights and thousands of travelers. Banking, insurance, and tax records can be found on mainframes, too. Utility companies keep their customer data on computers designed to be data servers. Programs run on personal computers (PCs) that provide people with application tools for bookkeeping, mathematical calculations, computer design packages, and image processing. Now you can access the Internet, perform professional word processing, or make your own greeting cards—thanks to powerful and easy to use programs. Microsoft Visual Studio 2005 is a program, too! Some programs run on customized microprocessors. These programs often control hardware such as robots, assembly line components, and motor controls for various tasks. The software developers (programmers) who write these programs, for the most part, take the same production steps.

Such procedures involve taking an idea, determining all the necessary behaviors and actions, writing the software instructions, building the program according to the structure required by the language, and testing it to ensure the end results are what were originally desired. The majority of this text involves learning how to write the C++ language software instructions—but it is important to keep the big picture in mind.

2.1

Programming Fundamentals

The first order of business for a programmer-wannabe is to understand that programming requires problem-solving skills. If you enjoy getting into the details of how to make something work, you have chosen the right career. Programmers are often presented with a problem or some sort of desired end result. (“Could you just make the computer do this?”) The programmer must understand the problem or the goal and then come up with a plan of how to achieve it. Recognize that this plan will not be problem-free.

Algorithm Design

algorithm

process or set of rules followed to solve a problem

Building and testing your software in steps is an essential habit for a software developer. The best approach is to use pencil and paper first, writing a plan for what you want to build. Something else must be mentioned while we're on the topic of software construction, and that is algorithm development. An **algorithm** is a process or set of rules or steps to follow for solving a problem. It is important that you create a set of steps that solves the problem before you sit down and start entering source code.

Let's look at a few examples. Suppose you had to write a program that reads lines of text from a data file. Your program should count the number of times the word *sheep* is found. How will you solve this problem? There are several approaches you can take. Figure 2-1 illustrates one. You can read each line of text into your program and then search the line letter by letter for the letter "s." If you find an "s," check the previous character to be sure it is a blank (or the "s" is the first letter in the line) and the next letter to see if it is an "h." If that letter is an "h," check the next for "e," and so on. If you find the five letters you need followed by a blank,

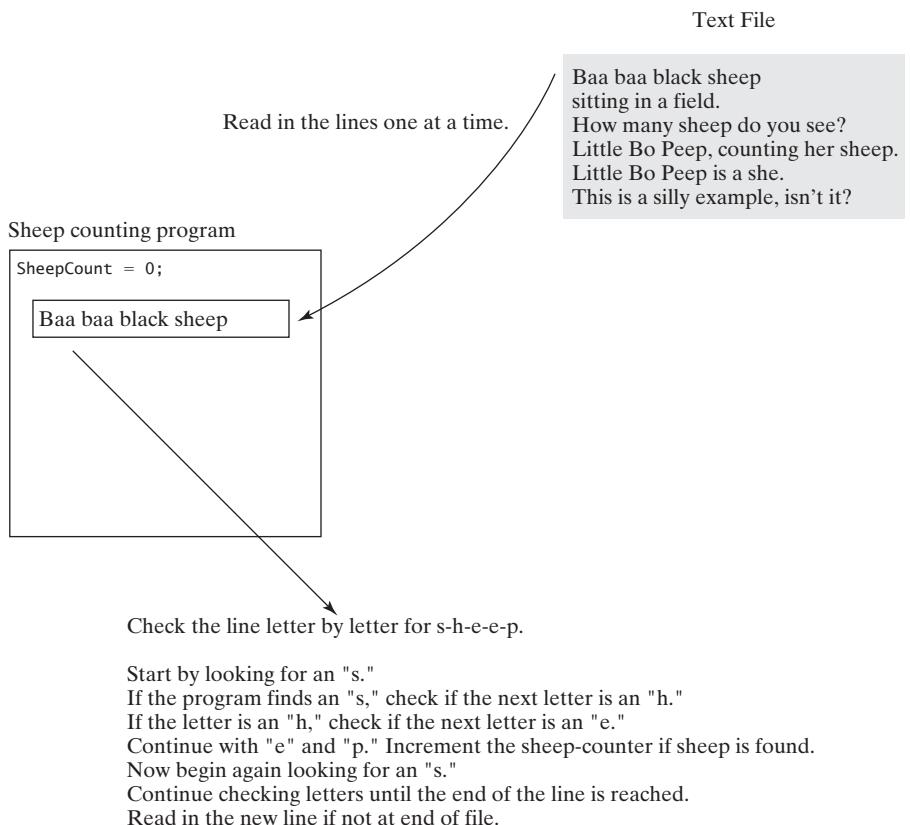


Figure 2-1

Algorithm for counting the word *sheep* in a line of text.

you have found the word “sheep.” Now you should increment a sheep-counter—that is, the count value that is holding the number of sheep you have found. This scheme of checking the letters is an algorithm. (There are, of course, other algorithms for counting sheep. Can you think of another one?)

What steps are needed for determining the total surface area of a cylinder? (Yes, math problems are everywhere in programming.) Pop the top of your favorite tasty beverage and refresh yourself as we examine how to determine how much wrapping paper it would take to cover the beverage can, including the top and bottom. Look at Figure 2-2. We need to know two pieces of information: the cylinder’s radius and height. We break the cylinder into three separate pieces: the top, side, and bottom portions. Using the radius and height, we can calculate the total surface area.

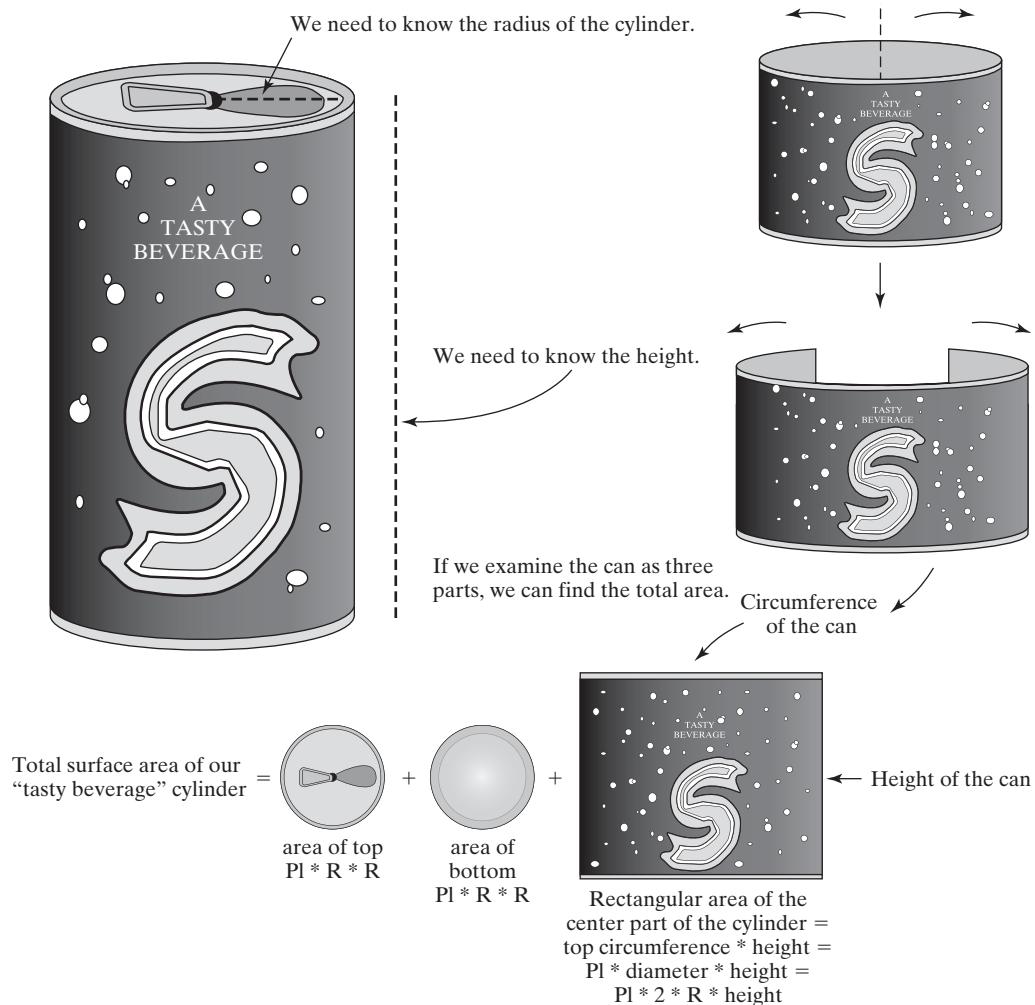


Figure 2-2

Algorithm for determining the surface area of a cylinder.

Steps to Programming Success

How do you become a good problem solver? Problem solving requires one to be methodical and careful. Let's create our own problem-solving algorithm with basic steps to aid us in understanding the problem and reaching a solution. As you begin to build software, you learn quickly that development is an iterative process. That is, it is best to build a small portion of the software, and make sure it works correctly (and has a good foundation) before you move on to building the next piece. As you build each piece, the software grows incrementally.

iterative and incremental development

process by which software is developed incrementally

For our problem-solving algorithm, we will model an ***iterative and incremental development*** approach. Simply put, you build your software incrementally (in steps) taking advantage of what you learned during the earlier versions of the software. You iterate through each step of the development process, evaluating the software design and functionality at each step.

One side note: something that you should always keep in mind as you build software—how will you know that your program is working correctly? Just because you are getting answers from the software doesn't mean that your answers are correct! Always think about testing your software as you design and build it.

Step 1. Read the Problem Statement Make sure you understand everything about the problem and what it is that you are being asked to do. You should know what the program inputs are and how they come into the program, as well as the program's outputs. Do not worry about *how* you are going to write the program, but do understand *what* the problem is and the desired results.

Step 2. Identify the Software Requirements What are the main program components you need to solve your problem? List them. Can you identify individual software components as we did in the ATM and car servicing examples in Chapter 1? What are the big steps that you must take to reach your programming goal? Identify the main pieces and an order in which they must be built. It is not necessary to lay out every detail at this stage, but you should know the major components of your program.

Step 3. Initial Step: Build a Subset of the Software Requirements Create a base version of your software that is functional enough so that you can see the key aspects of the program and how it is working. Before you sit at the computer and work on the coding, you should use pencil and paper to lay out the details and prepare test cases. You can write out a plan in diagrams, and for specific problems, develop flow diagrams or descriptions in short phrases known as pseudocode. As you work toward your solution, identify items that can be individually tested too.

Step 4. Evaluate the Base Version Once the base version is up and running, it is time for the developer to learn if this version meets the desired requirements or if a redesign is necessary. The base program should be tested. The developer can now go back and fix problems with the base version. If the base version is good, the programmer can make a list of the software tasks for the next version of the program, as well as develop testing methods for the next iteration of software.

Step 5. Iteration Step Implement the new tasks for the next version (or stage) of your software. The goal of the iteration is to be simple, straightforward, and

should support any redesign or new task. As you work on the software modifications and additions, you should document your source code explaining the important details.

Step 6. Evaluation Step Analyze the program now that the new tasks or features have been implemented and ensure that the software is working per the requirements list. When a programmer analyzes a program with a new portion of software, he should obtain feedback from a user, results from test cases, and determine how it meets the end goals. Be sure to test this new software, as well as the functionality from the previous iteration. (You do not want to have your new code break something in your previous code!) Make a list of the features for the next iteration of the software.

Repeat Steps 5 and 6 Continue to implement new tasks. Analyze and test until the program goals are complete.

You will find that the software projects you build using this approach move along smoothly and your frustration level (hopefully) remains low. Using an iterative process also provides nice stopping-points so that you are able to break a big job into small jobs, and tackle smaller problems one at a time. This gives you a different perspective on large tasks too—after all, how do you eat an elephant? (One bite at a time, of course.)

2.2

Terminology and Project Construction

Part of learning any new skill involves learning new terminology—computer scientists and software developers have their share of new terms. A search on the web using “software development terminology” will provide many complete glossaries for your reference. Table 2-1 presents several programming terms and phrases with their meaning that are used in this text.

Construction Steps

Computer systems that provide for C++ program construction may have different approaches for organizing and maintaining the programs on a system. Some systems, such as UNIX-based workstations or older versions of personal computer-based compilers, allow a one-file program to be compiled, linked, and run. Most programs actually have the source code separated into many files. For these programs, C++ and the computer systems require more information concerning the program and files. In the personal computer environment, Microsoft Visual C++ 2005 requires that the program be contained in a **project**. The project keeps track of all the files that are required for the program. In the UNIX world, there are software packages similar to Visual C++ 2005 (such as KDevelop), as well as a **make file** utility package that uses a convenient way to tell the compiler what files and libraries are needed.

It is time to study Appendix A, “Getting Started with Microsoft Visual C++ 2005 Express Edition.” This appendix covers installing this Microsoft IDE software for building C++ programs. Once the software is installed, we will create a project, write a short program, and compile, link, and run it. How to access the C++ Help is also presented.

project

the Visual C++ 2005 component that keeps track of all files needed for a program

TABLE 2-1

Common Programming Terms and Phrases

Term	Meaning
bug	General catchall word meaning the program is not running correctly.
class	A “job description” for a program component. The job description includes the tasks that the “worker” performs and associated data.
code	This term can refer to the textual-based phrases written in C++ that represent a program (or portion of a program). Code can refer to a single line, such as “a line of code” or the entire program. Also, it can refer to program file contents such as “machine code” or “executable code.”
compiler	Actual software (such as Microsoft Visual C++) that reads C++ statements. It checks that the statements are written with the correct syntax. The compiler produces object or machine code.
debugger	A tool in the software development package (such as Microsoft Visual C++) that allows the programmer to run the program one step at a time and to examine program portions. A debugger is used to track down bugs.
executable	The machine language file that the operating system reads. The operating system performs instructions based on the commands in this file. The executable file constitutes the program.
function	A discrete module or unit of code that performs specific tasks.
IDE	Integrated Development Environment. An IDE is a software development program (such as Visual Studio 2005 Express) that provides the developer complete tools for building software. IDEs contain an editor and tools for linking and executing code as well as access to Help files.
linker	Software that combines all the required files together and builds an executable file.
object	In object-oriented programming, an instance of a class.
object code	File produced by a compiler. The source code file is translated into machine language.
RAM	Random Access Memory. Actual computer chips that contains storage areas that are directly accessed by the computer operating system and programs. As a program executes, its data items are stored in RAM.
source code	The text-based file containing C++ statements. The source code is read by the compiler.
syntax	The correct way in which the language’s words and symbols are put together so that they have meaning to the C++ language. It may be thought of as the “grammar” and “punctuation” rules for the language.

2.3**General Format of a C++ Program**

A C++ program consists of several basic components, whether the program consists of a few lines or is large and complicated. Before we dissect several programs, let's step back and introduce two important concepts: case sensitivity and functions. C++ is ***case sensitive***. C++ recognizes the difference between UP-PERCASE and lowercase letters. The language will view the names "total," "Total," and "TOTAL" as three separate things. As another example, the operating system knows to look for the "main" function, and the C++ compiler automatically recognizes a function named main. However, the compiler does not know what a Main or MAIN function is (unless the programmer creates user-defined functions).

A ***function*** is a block of code that has a specific format, with an entrance point, and one (or many) exit points. All functions have a specific name, a means to pass data into it and obtain data from it. Functions allow us to build code blocks that do program tasks. All activities in a C++ program are performed inside functions. In Chapters 2 and 3, we build programs that consist of only one function. All C++ programs need to have a starting point, and for us, it is the ***main function***. Later we will build large programs using many functions (and several source code files), but to get us started, let's do Hello World.¹

case sensitive

C++ recognizes upper- and lowercase letters are different

function

a discrete module or unit of code that performs specific tasks

main function

a function named "main," that is the starting program for our C++ programs.

Hello World! Program

In our first program, we write **Hello World!** to the screen. This program consists of one function named main, and other lines of code are necessary, too. This program is also presented in Appendix A, "Getting Started with Visual C++ 2005 Express Edition." The source code for our Hello World! program is shown in Program 2-1.

Program 2-1

```

1 //Getting Started with Hello World.
2 //This is our first program.
3
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     cout << "Hello World!  \n";
10    return 0;
11 }
```

Program Output

Hello World!



¹Hello World! is the first program in Kernighan and Ritchie's *The C Programming Language*. In their honor, it will be our first program as well.

Comments

The first two lines in the example are comment lines.

```
//Getting started with Hello World.  
//This is our first program.
```

comment

lines in source code, ignored by the compiler, in which programmers may write information concerning the file or program

Comment lines are written into the source file by the programmer to relay information concerning the code—information such as titles, file data, or explanations about what the software is doing. The compiler ignores all comment lines in a program. There are two ways to write comments in C++.

```
//This is one way to write a comment.  
//The compiler ignores everything from the two forward slashes  
//to the end of the line.
```

These // comments are especially convenient for one-line comments. A second way to write a comment is:

```
/* Here is another way to write a comment.  
The compiler ignores everything between the  
beginning slash-star and the ending star-slash. */
```

This comment style is convenient for writing multiple-line comments because it needs only the /* at the start and */ at the end of the comment lines.

Preprocessor Directives

The next line

```
#include <iostream>
```

preprocessor directive

lines, such as #include or #define lines, in the source code that give the compiler instructions

#include

preprocessor directive telling the compiler to read the given file

is known as a **preprocessor directive**. The directive gives the compiler instructions. The most common type of directive includes additional statements in the program. The preprocessor directive statements are usually at the top of the file. In Hello World! we need to include the iostream library because it contains the necessary program tools to write text to the screen. The information is located in a header file named iostream (the actual file is named iostream.h.) The **#include** statement tells the compiler that this short program uses one or more of the iostream tools. (All of the C++ preprocessor directive statements are shown in Appendix H, “Multi-file Programs.”)

The C++ language provides many libraries with many tools (pre-defined classes and functions) for programmers to use, as did the C language. If you wish to use one of the standard C or C++ functions (such as generating a random number, taking a square root, or writing data to a file) you have to include the name of the appropriate library. Take a moment to look at all of the C++ libraries. The libraries are shown on the inside front cover of this text.

The top portion of this list shows you the libraries that we use in this text, with a brief description of their purpose. The bottom portion shows you the complete list of C++ libraries. You may have seen C libraries that are referred to with a “.h” extension, such as <math.h> or <time.h>. All of the C libraries have been incorporated into the C++ libraries (i.e., <cmath> and <ctime>, etc.) The C++ libraries are

written without the “.h,” such as `<iostream>` and `<iomanip>`. If you were writing C programs, to include the necessary libraries you had to type the “.h” in the `#include` statement. If you are using a C++ library, you do not type the “.h” in the statement.

In your C++ program, If you wish to take a square root of a number and then write it to the screen showing four decimal places of accuracy you need the `sqrt()` function—which is in C++’s `<cmath>` library. To write to the screen, you need `iostream`’s `cout`. There are formatting commands in `iomanip`. Your program needs to include these three libraries:

```
#include <cmath>           //used for sqrt() function
#include <iostream>         //used for cout to write output to the screen
#include <iomanip>          //for output formatting utilities
```

The `using namespace std` statement

The line following the include statements is

```
using namespace std;
```

This tells the compiler that your program is using the standard (std) C++ namespace which includes the C++ libraries. The namespace idea is simple—even though it has a confusing name. Namespace is used to help organize the program and avoid problems with program components having the same name. It is possible to have a large program (or even a small one) where programmer(s) have named functions with the same name. Imagine how confused the compiler would be if it found two identically named functions! The designers of C++ foresaw this problem and gave programmers the ability to organize their code into namespace regions. Don’t worry. We will not be writing our own namespaces. We need to have the “`using namespace std;`” statement after our `#include` statements, and we are good to go.

The `main` Function

The line

```
int main()
{
```

is the starting point for a C++ program. When the program begins to run, the operating system (which runs the program) starts performing the actions directed by the C++ statements. Our C++ programs will contain a `main` function, and the operating system knows to look for it. Think of the `main` function as the “program director” giving orders to other program components via calls to functions (or through the use of objects). The lion’s share of the program is not performed in the `main` function.

Function Header Line

It is never too early to begin learning function details, so let’s spend a bit more time examining the first line of our `main` function:

```
int main()
```

function header line

the first line of any function in C/C++

The first line of any function is known as the ***function header line***. It specifies the function's name and its input and output information. The general format for a function header line is:

```
return_type function_name(input parameter list)
```

where *return_type* (*int* in Hello World!) is the type of data that the function passes to whomever calls it. The input parameter list is the list of data that is passed into the function. In C++ the *main* function returns an integer value to the calling process (the operating system). We use this standard function header line for our programs. (Chapter 4 contains more details on function header lines.)

All C++ functions have an opening brace { after the function header line and at the end of the function is a closing brace }. These braces enclose the statements of the function. Braces are used extensively in C++.

C++ Statements

A C++ statement is part of the program that issues a command to be executed. It specifies an action that must occur. Our Hello World! program has two C++ statements:

```
cout << "Hello World!  \n";
return 0;
```

cout

the object from
<iostream> we use to
direct output to the
screen

This ***cout*** statement (pronounced cee-out) is an object that we use to direct output to the screen. We use the “<<” operator to send data that will be shown in the console window (output window). The *cout* statement here is written according to the C++ language rules and ends with a semicolon; many, but not all, statements in C++ end in a semicolon. The “return 0;” statement is located at the end of the *main* function. The return type for the *main* function is an *int* (integer). The *main* function uses this return statement to pass a zero to the operating system at the conclusion of the program.

How's the Weather?

Our second sample program (Program 2-2) contains several *cout* statements and both types of comment styles. The program writes three statements to the screen. The \n is an escape sequence, which causes the output to be printed to the next line on the screen. (We will cover many of the output formatting statements in this Chapter, and summarize in Section 2.8)

Program 2-2

```
1  /* How's the weather?
2
3  This program writes a few weather-related
4  statements to the screen.  */
5
6  #include <iostream>
```

```
7  using namespace std;  
8  
9  int main()  
10 {  
11     //Weather Information  
12     cout << "\n It is cloudy today.";  
13     cout << "\n Maybe it will rain.";  
14     cout << "\n I like sunny, warm weather! \n";  
15     return 0;  
16 }
```

Program Output

It is cloudy today.
Maybe it will rain.
I like sunny, warm weather!

Figure 2-3 illustrates the source code and the screen output. Note that the comments are seen only in the source code, and the text information in the *cout* statements are seen in the program output.

Whitespace Characters and Flexible Style in C++

Whitespace characters are defined to be spaces, carriage returns, line feeds (the Enter key), tabs, vertical tabs, and form feeds. For the most part, the C++ compiler ignores whitespace characters. Why is this important to the C++ programmer? It means that the compiler allows the programmer to write his or her code with the use of a wide variety of styles and formats. (The compiler is very particular about the syntax, of course, but style is up to the programmer.) Are you *stylish*?

Unlike the COBOL or FORTRAN languages, which have restrictions about what data must be in which column and restrictions for the length of lines, C++ statements can be written in virtually any format. For example, our Hello World! program could be written on one line, as shown in Program 2-3. The How's the Weather? program could be entered in the file like Program 2-4.

whitespace

spaces (blanks), tabs, line feeds, enter key, control characters, vertical tabs, and form feed characters

Program 2-3

```
1  //Our Hello World program written in one line.  
2  
3  #include <iostream>  
4  using namespace std;  
5  
6  int main(){ cout << "Hello World! \n"; return 0;}
```

Program Output

Hello World!

```
//Source code: weather.cpp
/* How's the weather?

This program writes a few weather-related statements to the screen. */

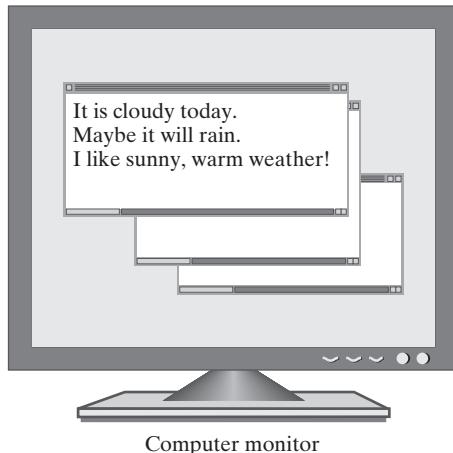
#include <iostream>
using namespace std;

int main()
{
    // Weather information

    cout << "\n It is cloudy today.";
    cout << "\n Maybe it will rain.";
    cout << "\n I like sunny, warm weather!";
    return 0;
}
```

Comment lines
Preprocessor directive
Function header line
Single line comment
C++ statements

Program output (seen on the computer screen in a window)



Computer monitor

Figure 2-3

How's the Weather? Source code and program (screen) output.

Program 2-4

```
1  /*The Weather program re-written.
2  In this program we'll write a few weather
3  statements to the screen. */
4
5  #include <iostream>
6  using namespace std;
7
8  int
9  main() { // Weather information
10   cout <<
11   "\n It is cloudy today. ";
```



```

12 cout <<
13 "\n Maybe it will rain.";
14 cout <<
15 "\n I like sunny, warm weather! \n";
16 return 0;
17 }
```

Program Output

It is cloudy today.
 Maybe it will rain.
 I like sunny, warm weather!

The second versions of the Hello World! and How's the Weather? programs compile without error because the C++ compiler does not require the programmer to start each statement on a new line. Both programs produce the same output as their initial versions. However, it is obvious that the source code for both is difficult to read.

Well-written comments are quite significant! Comments make the source code easier to read, and they document what the code is doing. Without comments, the programmer has difficulty remembering what he or she was doing, and it is almost impossible for a new person to read and understand the code at a later date. Conversely, too many comments can obscure the code.

Language Syntax and Compiler Errors

The flexibility feature for your C++ style does not mean that you may write C++ statements in just any old way. Although spacing does not matter, C++ must be written with correct **syntax**. What do we mean by a grammar or a syntax rule? Let's first examine how spoken languages have grammatical rules. Table 2-2 shows four written languages asking the question "How are you?" Table 2-3 illustrates computer programming language instructions for writing the question, "How are you?"

A syntax rule is a rule for writing the language. If C++ requires the word `#include` to be in lowercase letters, then a syntax error is found if you capitalized the first letter (`#Include`) or use all caps (`#INCLUDE`). For example, this source code contains several syntax errors and does not compile. Examine the code and see if you can spot the errors the compiler will catch.

syntax

grammatical rules required by the language

TABLE 2-2
 Spoken Languages Have Grammar Rules

Language	How are you?
English	How are you?
German	Wie geht's Dir?
French	Comment va tu?
Spanish	¿Como estas?

Table 2-3
Computer Programming Languages Also Have Grammar Rules

Language	How are you?
BASIC	<i>PRINT "How are you?"</i>
FORTRAN	<i>WRITE(6,35)</i> <i>35 FORMAT('HOW ARE YOU?')</i>
C	<i>printf("How are you?");</i>
C++	<i>cout<<"How are you?";</i>
Java	<i>System.out.println("How are you?");</i>

Program 2-1 With Compiler Errors

```

1 //Our Hello World! program with syntax errors
2 //This code will not compile.
3
4 #INCLUDE <iostream>           //#include cannot be all capital letters
5 using namespace standard;    //C++ knows "std", not standard
6
7 int Main ()                  // main must be all lowercase
8 {
9     Cout << "Hello World!"; //cout must be lowercase
10    return 0                 //missing ;
11 }
```



Compiler Error

Hello.cpp(4) : fatal error C1021: invalid preprocessor command 'INCLUDE'

Although this code has many compiler errors, the compilers do not understand the incorrectly written `#include` statement on Line 4. Compilation stops at the preprocessor line and issues an error showing the line number.

When we fix the `#include` statement, our code looks like this and the compiler now sees four errors:

Program 2-1 With Compiler Errors

```

1 //Our Hello World! program with syntax errors
2 //This code will not compile.
3
4 #include <iostream>           // #include now correct
5 using namespace standard;    //C++ knows "std", not standard
6
7 int Main ()                  // main must be all lowercase
8 {
9     Cout << "Hello World!"; //cout must be lowercase
10    return 0                 //missing ;
11 }
```

Compiler Errors

```
Hello.cpp(5) : error C2871: 'standard' : does not exist or is not a namespace
Hello.cpp(9) : error C2065: 'Cout' : undeclared identifier
Hello.cpp(9) : error C2297: '<<' : illegal, right operand has type 'char [13]'
Hello.cpp(11) : error C2143: syntax error : missing ';' before '}'
```

Compilers do the best they can in determining syntax errors. Often just a few errors in the code will generate many compiler errors! Don't be discouraged. Just start at the top of the list and work your way to the end. In our list above, we see that the misspelled “*Cout*” statement is not recognized by the compiler, and hence it doesn't understand the “`<<`” on the following line. Sometimes the error is on the line above the one shown in the error list. The error being reported on Line 11 is due to the lack of a “`;`” on the previous line. You will soon learn, however, the compilers cannot read your mind. *The programmer must take the time to learn the correct syntax for C++ and thus be able to spot compiler errors quickly.*

Two Helpful Hints The Microsoft Visual C++ compilers always shows comments in one color (such as green), words that are reserved by the language in another (such as blue), and statements in black. (These colors can be customized by the programmer.) If you are entering a reserved word (keyword or preprocessor directive), such as `#include`, it should be in its color. If the word is not in blue, double-check the way you have entered it. Second, the error is often located on the line above the referenced line. If you get a compiler error and the referenced line appears to be correct, look above that line for the error.

C++ Keywords

The C++ language has many reserved words known as **keywords**. These keywords have specific meaning for the language and may not be used by the programmer as names for variables or functions. Each keyword has specific syntax and actions associated with it. For example, the two keywords `switch` and `case` might be popular variable names in electronics or inventory programs. C++ reserves these two words for its own use (`switch` is a conditional statement and `case` is a label). The C language contained thirty-one keywords. The C++ language expanded that list to sixty-three. Appendix B, “C++ Keyword Dictionary,” shows all of the C++ keywords as well as short usage examples and a text reference. Table 2-4 shows the sixty-three keywords in C++.

Keywords

reserved words in the language that have specific syntax and actions

Good Style

As mentioned before, it is necessary for the programming student to develop a readable style when writing source code. This legible style includes writing descriptive comments, choosing appropriate variable names, and indenting the source code from the start of a programming project. Always assume that someone else needs to read your code and figure out what you are doing. Students often write their code and “make it pretty” later. Time spent making it pretty initially is time saved when trying to debug compiler errors.

TABLE 2-4
C++ Keywords

<i>asm</i>	<i>auto</i>	<i>bool</i>	<i>break</i>	<i>case</i>	<i>catch</i>
<i>char</i>	<i>class</i>	<i>const</i>	<i>const_cast</i>	<i>continue</i>	<i>default</i>
<i>delete</i>	<i>do</i>	<i>double</i>	<i>dynamic_cast</i>	<i>else</i>	<i>enum</i>
<i>explicit</i>	<i>export</i>	<i>extern</i>	<i>false</i>	<i>float</i>	<i>for</i>
<i>friend</i>	<i>goto</i>	<i>if</i>	<i>inline</i>	<i>int</i>	<i>long</i>
<i>mutable</i>	<i>namespace</i>	<i>new</i>	<i>operator</i>	<i>private</i>	<i>protected</i>
<i>public</i>	<i>register</i>	<i>return</i>	<i>reinterpret_cast</i>	<i>short</i>	<i>signed</i>
<i>sizeof</i>	<i>static</i>	<i>static_cast</i>	<i>struct</i>	<i>switch</i>	<i>template</i>
<i>this</i>	<i>throw</i>	<i>true</i>	<i>try</i>	<i>typedef</i>	<i>typeid</i>
<i>typename</i>	<i>union</i>	<i>unsigned</i>	<i>using</i>	<i>virtual</i>	<i>void</i>
<i>volatile</i>	<i>while</i>	<i>wchar_t</i>			

Love My Style

Programs 2-5 and 2-6 illustrate how important good style can be when trying to read source code. Both of these programs compile with no error and write the same output to the screen. Which program would you rather read?

In the first Love My Style program (Program 2-5), the comments are jammed against the C++ statements, making both the statement and comment hard to see. Also, the comments do not tell the reader anything helpful about the program. In the second program, the comments contain more information about the program and obvious or redundant comments are omitted. Throughout this text, we will work on code and commenting style techniques.

Program 2-5



```

1  /*Coding style you love to hate.
2  This program writes style remarks to the screen.*/
3  #include <iostream>//iostream library
4  using namespace std;
5
6  int main()/* cout output statements*/
7  cout<<"\nI am a programmer.";//write out I am a programmer
8  cout<<"\nThis is great code.";
9  cout<<"\nDo you like my style?"
10 << "\nI am all finished now. \n";
11 return 0;}//end of program

```

Program Output

I am a programmer.
This is great code.
Do you like my style?
I am all finished now.

Program 2-6

```

1  /* Coding style you love to love.
2      This program writes programming comments to the screen. */

```

```
3  
4 #include <iostream>           //needed for cout  
5 using namespace std;  
6  
7 int main()  
8 {  
9     cout<<"\nI am a programmer.";    //write style comments to screen  
10    cout<<"\nThis is great code.";  
11    cout<<"\nDo you like my style? \nI am all finished now. \n";  
12    return 0;  
13 }
```



Helpful Hint If you are writing C++ source code in an Integrated Development Environment (IDE), such as Microsoft Visual C++ 2005 Express, KDevelop, or Eclipse, the editors automatically indent your source code lines in a C++ friendly manner. Allow these editors to indent your code so that the code is more readable. The indentation helps you line up your braces, since every open brace { needs a close brace }.

One Last Comment on Comments

Your comments should explain your thought process and logic. The following comment

```
#include <iostream> //include the iostream library
```

does not tell the reader why the library is included, and it is too close to the end of the `#include` statement. We become spoiled by the statement and keyword color-coding in the various C++ IDEs, but once printed in black and white, it is hard to tell where the statement ends and the comment begins. In the line below, the comment is offset from the source line and explains why this library is needed in the program:

```
#include <iostream>           //needed for cout statement
```

Ensure that your comments help the reader understand what you are writing instead of wasting space and effort explaining the obvious.

2.4

Program Data and Data Types

Imagine that you and your family throw a party and cook a huge dinner for 15 of your friends and family. After your guests depart, you stand in the kitchen amid the mess. The first order of business is to put away the remaining food so that it doesn't spoil. How do you go about doing this task?

You notice that there is a little bit of salsa in its bowl and a few chips on the plate. The veggie tray and dip was not finished and there are several pieces of barbecue chicken on the plate. There is one lonely piece of pie in the pan, four

ears of corn on a plate, and there is about a quart of the homemade ice cream sitting on the counter. As you size up your leftovers, you need to decide what sort of containers you need to store your food. The salsa and veggie dip need small containers with lids. The veggies and uneaten chips can go into sandwich bags. The ears of corn can be wrapped individually, and the chicken requires a large container with a lid. The ice cream requires a larger bowl-like container that can be sealed. Pretend that your containers are all opaque, so you take a marker and label each item: chicken, veggie dip, corn, salsa, etc (You get to take care of that last piece of pie!)

What in the world are we up to talking about storing leftovers? You may also ask an important question: “What kind of pie are we talking about?” No, seriously. This business of examining the food and determining the correct container is analogous to what C++ programmers need to do when designing a program. You, the programmer, need to determine how many “containers” are required for your program data, and select the appropriate type of container. You must also give each container its own name.

As you are designing your program and thinking through the logical steps, you must also ask yourself what types of data the program will handle. Will the program need “containers” for numeric data or textual (character) data? How many different containers are required in the program? What level of accuracy (number of decimal places) is required? A bookkeeping program needs only two or three decimal places; on the other hand, an airborne radar system requires twenty decimal places.

data type

type of “container” that holds program data, including *int*, *float*, *double*, *char*, and *bool*

array of chars

a group or list of characters referred to with one name
(See Chapter 6)

string

a C++ data type (a class) that is used for textual data

char

data type for variables that contain a single character

float

data type for variables containing up to five digits of decimal precision

double

data type for variables containing up to ten digits of decimal precision

int

data type for variables containing whole numbers

Data Types in C++

If you are writing a banking program, it probably has a variety of data including name and address information; balance, deposit, and withdraw amounts; account numbers; personal identification numbers; and transaction counters. The money data need to be numeric—the name and address information must be textual data. It is important to have decimal-place accuracy for the money (to keep track of pennies), but our transaction counter can be a whole number.

The C++ language provides a variety of data types for the programmer. A **data type** is a type of “container” that can hold a specific kind of program data. For each piece of data in your program, you must specify the appropriate type of container that must be used. For example, a customer’s name needs to be stored as characters. This is accomplished using either an **array of chars** or as a **string**. Money information must be stored in either a **float** or a **double** (to maintain the decimal portion), and a transaction number can be stored as a whole number, which is referred to as an **int**. Table 2-5 lists the basic data types provided by C++.

Containers = Data Types, Labels = Variable Names

A programmer must designate the type of data container, data type, and name the container’s name, variable, for each piece of data in the program. The data that is placed in the variable is referred to as the variable’s value. When the program executes, physical memory in the computer system is set aside for each piece of data. (In our kitchen analogy, you may think of the refrigerator,

TABLE 2-5

Basic Data Types in C++

Data Type	Name	The Data It Contains	Example
<i>char</i>	char or character	A single character	a
<i>int</i>	integer	A whole number (no decimal point)	43
<i>float</i>	float or floating point	A number with six to seven digits of precision	14.937453
<i>double</i>	double	A number with thirteen to fourteen digits of precision	3.14159265294753
<i>void</i>	void	Empty or nothing; specifies function as returning no values	(is presented later)
<i>bool</i> ^a	boolean	Stores values of true or false	true
<i>wchar_t</i> ^a	wide character	Holds wide characters (16 bits)	Japanese character

^aNote: Not defined in the C language.

freezer, and cabinets as storage locations in your kitchen.) Selecting the appropriate type of container for your leftovers is analogous to specifying a data type in your program. (Remember that containers hold different things.) Placing the label on the container and giving the variable a name serve a similar purpose because you know the name of the container variable.

To understand the different data types and range of values they may hold, we need to introduce the concept of a *byte*. The byte is the basic unit of computer memory. The byte consists of eight *bits*. A bit is a unit of data that exists in one of two states. We often refer to a bit as being either a 1 or a 0. Check Appendix E, “Bits, Bytes, Memory, and Hexadecimal Notation,” for further discussion.

The ISO Standard C++ language specifies the minimum number of bytes of storage that each data type must reserve and use for storing values. These reserved bytes dictate how big a value may be stored. (Back to the kitchen analogy: we can’t stuff the leftover chicken into a sandwich bag!) The size of the container dictates the range of data in which it can hold.

Figure 2-4 illustrates how the number of bits dictates the number of unique bit-combinations. For a value to be stored, it must be represented by a unique bit pattern. A 1-byte storage container can have only 2^8 , or 256, unique bit combinations. Figure 2-5 takes this concept a bit further. If a data type reserves 2 bytes, then there are 16 bits for storage. Four bytes result in 32 bits of storage space. Examining the possible combinations shows us how reserving more bytes results in a larger number of combinations, and hence how larger values (a wider range of values) may be stored.

byte

the basic unit of computer memory consisting of eight bits

bit

a unit of data that exists in one of two states, such as 1 or 0, on or off

Data Type Modifiers

C++ allows the use of modifiers to create many more data types (or specialized containers). The modifiers include the terms (keywords in C++) *short*, *long*, and *unsigned*. The complete list of data types is shown in Table 2-6. The ISO C++

Bits	Possible Bit Combinations	Unique Values
1	0	0 The number of unique values can be calculated by 2^n , where n = number of bits.
	1	1 $2^1 = 2$ combinations
2	00	0
	01	1 $2^2 = 4$ combinations
	10	2
	11	3
3	000	0
	001	1
	010	2 $2^3 = 8$ combinations
	011	3
	100	4
	101	5
	110	6
	111	7
8	00000000	0
	00000001	1 $2^8 = 256$ combinations
	:	:
	11111111	255

Figure 2-4

The number of bits dictate the number of unique combinations and hence unique (different) values.

Bytes	Possible Combinations	Signed Range	Unsigned Range
1	8 256	-128 to 127	0 to 255
2	16 65,536	-32,768 to 32,767	0 to 65,535
4	32 4,294,967,296	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295

$2^8 = 256$ combinations

$2^{16} = 65,536$ combinations

$2^{32} = 4,294,967,296$ combinations

Figure 2-5

More bytes for the data type means that its variables hold a larger range of values.

TABLE 2-6

Data Types Defined by the ANSI/ISO C Standard

Keyword	Typical Bytes of Memory	Precision Range
<i>char</i>	1	–128 to 127
<i>unsigned char</i>	1	0 to 255
<i>signed char</i>	1	–128 to 127
<i>int</i>	4	–2,147,483,648 to 2,147,483,647
<i>short int</i>	2	–32,768 to 32,767
<i>unsigned short int</i>	2	0 to 65,535
<i>unsigned int</i>	4	0 to 4,294,967,295
<i>long int</i>	4	–2,147,483,648 to 2,147,483,647
<i>unsigned long int</i>	4	0 to 4,294,967,295
<i>float</i>	4	6 digits, i.e., 0.xxxxxx 3.4 E ± 38 (7 digits in Visual C++)
<i>double</i>	8	10 digits, i.e., 0xxxxxxxxx 1.7 E ± 308 (15 digits in Visual C++)
<i>long double</i>	10	10 digits, i.e., 0xxxxxxxxx 1.2 E ± 4932 (19 digits in Visual C++)

Standard does not specify the exact amount of storage that the C++ program must use for the various data types, but it does specify the minimum. The programmer must be aware that sizes and ranges may vary.

Troubleshooting: How Big Is the Integer?

The ANSI/ISO Standard specifies that data types must meet certain requirements but does not specify exact ranges. In the “old days” the integer “container” could vary between 2-bytes and 4-bytes. On systems with 16-bit architecture (such as older PCs), integers typically had a 16-bit (2-byte) integer. The 2-byte *int* meant that an integer variable could hold a value with a range from –32,768 to 32,767. Today computers have 32-bit or 64-bit architecture, and the integer is 4 bytes, meaning that the *int* can hold values from –2 billion to +2 billion. (See the exact range in Table 2-6.)

A new programming arena involves writing code for mobile devices (typically in the Java language) where programming space may be limited. Programmers may opt to use short ints (2-byte) to save programming space. But remember! If you are writing code using a 2-byte *int*, the value range is about +/-32,767. If your program needs integers values larger than 32,767 (such as counting the attendance at an NFL football game), the program needs to be using a 4-byte integer. To illustrate how important it is to be aware of the size and value range of your system’s integer, read the system failure that occurred with the Ariane 5.



Explosion of the Ariane 5* On June 4, 1996, an unmanned Ariane 5 rocket launched by the European Space Agency exploded just forty seconds after liftoff. The rocket was on its first voyage, after a decade of development costing \$7 billion. The destroyed rocket and its cargo were valued at \$500 million. A board of inquiry investigated the causes of the explosion and in two weeks issued a report. The cause of the failure was a software error in the inertial reference system. Specifically, a 64-bit floating point number (i.e., a double) relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16-bit signed integer. The number was larger than 32,768, the largest integer that can be stored in a 16-bit signed integer, and thus the conversion failed.

*This Ariane 5 launch is a classic study of a system failure due to software. The excerpt is adapted from the Institute for Mathematics and its Applications web site (<http://www.ima.umn.edu/~arnold/disasters/ariane.html>). The IMA web site provides a link to the full failure report, *The ARIANE 5 Flight 501 Failure Report* by the inquiry board. A search on the Internet for "Ariane 5 Failure" will result in many references.

2.5 Variable Declaration in C++

variable

memory location reserved by the program for storing program data

value

actual data stored in the variable

variable declaration

C++ program statement dictating the type and name for a variable

A **variable** is an actual location in memory that has been set aside for use by the program and it is referenced by a specific name. A variable contains a data **value** that may be modified by the program. The program must declare a variable by stating the type of data it is to contain (*int*, *float*, *double*, etc.) and give the variable a name. As you begin writing your code, think of the memory locations you will need. When you declare a variable, the computer will reserve a memory location for your variable. This declaration is performed only once in a function, and then the variable is ready to be used as often as needed. (Note: Variable scope is discussed in Chapter 4, which covers global, local, and static variable properties.)

A **variable declaration** statement must have a data type and a variable name. The basic format is

```
data_type variable_name;
```

For example, in our banking program the money, count, and check number variables could be declared as follows:

```
//Declaration of variables for banking program
float balance;
float deposit;
float withdraw;
int transaction_count;
int check_number;
```

In these statements we are setting up three floating point variables—balance, deposit, and withdraw—and two integer variables—transaction_count and check_number. When this program runs, five separate memory locations are reserved, one location

for each variable. It is valid to have several variables and one data type on one line, such as:

```
//Declaration of variables for banking program
float balance, deposit, withdraw;
int transaction_count, check_number;
```

Naming Rules in C++

In C++, an **identifier** is the name of a user-defined object, a variable, a function or a label. **Identifier naming rules** are listed below. It is always a great idea to use descriptive names for your program items.

- Names may contain letters (A to Z, a to z), numbers (0 to 9), or underscores (_).
- The first character must be a letter or an underscore.
- Names cannot contain any symbols, such as ~ ! @ # \$ % ^ & * () - "+ = \ / ', nor can they have any spaces.
- Keywords cannot be used as variable names.
- Identifiers may be any length, but only 1,024 characters are significant.

Table 2-7 shows examples of valid and invalid variable names.

Where Can You Declare Variables?

Variables in C++ can be declared in three places in a program: inside a function, outside a function, and in a function header line. Where the declaration occurs dictates what parts of the program can see and have access to the variable values. This access is known as the scope of a variable and is discussed fully in Chapter 4. For now, we will declare our variables in the function header line or inside the function.

TABLE 2-7

Valid and Invalid Variable Names

identifier

the name of a variable, label, function, or object that the programmer defines

identifier naming rules

uses A-Z, a-z, 0-9, _;
can't start with 0-9;
no spaces, no keywords, no symbols

Variable Name	Valid or Invalid	If Invalid, Why?
<i>balance</i>	Valid	(Not applicable)
<i>transaction amount</i>	Invalid	Contains a space
<i>convert_2_#s</i>	Invalid	No symbols like #
<i>MyMoney</i>	Valid	(Not applicable)
<i>case</i>	Invalid	case is a C++ keyword
<i>Case</i>	Valid	(Not applicable)
<i>4_temperature</i>	Invalid	Cannot start with number
<i>auto</i>	Invalid	Cannot be a keyword
<i>My_auto</i>	Valid	(Not applicable)

C++ requires that a variable be declared before it is used, which makes sense because you need to have a storage container set up before you can store a value. Also, C++ allows variable declaration within conditional statements—but these variables are visible only to the code within the block of statements. More on this topic in the next chapter.

2.6 Operators in C++

operators

symbols that represent certain instructions or commands in C++

The C++ language has many operators, which provides wonderful flexibility for the programmer. (A complete table of C++ operators appears in Appendix C.) **Operators** are symbols that represent certain instructions or commands. A simple addition example shows the arithmetic operator `+` and the assignment operator `=`:

```
//Operator Example Addition
sum = x + y;
```

A second example shows temperature conversion using operators:

```
//Operator Example Temp Conversion
F_temp = 9.0/5.0 * C_temp + 32.0;
```

This statement is interpreted as “first divide 9 by 5, then multiply by `C_temp` and add 32, and then assign the result into `F_temp`.” It is important to note that the assignment operator (`=`) is on the left-hand side of the statement, and the multiplication/addition is on the right. Do not write the equation like this, as it will not compile.

```
//INCORRECTLY WRITTEN!
x + y = sum;
9.0/5.0 * C_temp + 32.0 = F_temp;
```

Road Trip Calculation Program

Want to get away? Road trips can be a lot of fun, especially if you enjoy time, rate, and distance problems from your algebra class. In Program 2-7 we see how to declare variables in C++ and then calculate a few fun facts for our road trip. Here we travel at an average of 68 mph, for four days, driving 7.5 hours each day. Note the various ways we use to write declaration statements and assign values to the variables.

Program 2-7

```
1 //This program calculates the number of miles and hours
2 //traveled based on average speed and days driven.
3
4 #include <iostream>           //for cout
5 using namespace std;
6
7
```

```
8 int main()
9 {
10    //declare our variables here
11    //we can assign the values when declared
12
13    double speed = 68.0;           //average rate mph
14    int daysOnRoad = 4;           //driving days
15    double hrsEachDay = 7.5;      //hours driven per day
16
17    double dailyMiles, totalMiles, totalHoursDriven;
18
19    //calculate miles driven each day hours x speed
20    dailyMiles = hrsEachDay * speed;
21
22    //calculate total hours by days x hours
23    totalHoursDriven = daysOnRoad * hrsEachDay;
24
25    //total miles is total hours x average rate
26    //(or could have used daily miles x days driven)
27    totalMiles = totalHoursDriven * speed;
28
29    cout << "\n Driving Trip Summary ";
30    cout << "\n " << daysOnRoad << " days on road driving " << speed
31          << " mph for " << hrsEachDay << " hours each day";
32    cout << "\n\n Daily miles: " << dailyMiles;
33    cout << "\n Total driving hours: " << totalHoursDriven;
34    cout << "\n Total miles driven: " << totalMiles << "\n\n";
35
36    return 0;
37 }
```



Output

Driving Trip Summary
4 days on road driving 68 mph for 7.5 hours each day

Daily miles: 510
Total driving hours: 30
Total miles driven: 2040

C++ allows the programmer to be flexible in writing declaration and assignment statements. In this program we have declared speed, daysOnRoad, and hrsEachDay on separate lines, while assigning their values to them. Another way to accomplish the same thing would be to write the code like this:

```
//declare first
double speed, hrsEachDay;           //average rate in mph, and hrs per day
int daysOnRoad;                     //driving days
//then assign
```

```
speed = 68.0
hrsEachDay = 7.5;
dayOnRoad = 4;
```

A third way to write these lines have the declaration and assignment performed in one line, like this:

```
//declare rate in mph and hrs per day variables
double speed = 68.0, hrsEachDay = 7.5;
int daysOnRoad = 4;
```

You may ask the question, “Which is the best way?” The answer to that is up to you. It is important that your code is readable and that you write comments explaining what the variables are, if their tasks aren’t obvious.

Obtaining Program Data from the Keyboard

cin

found in the `<iostream>` library, `cin` `>>` is used to read program data from the keyboard

Up to now, our C++ programs have performed simple calculations and written data to the screen using the `cout` object. To make our programs more interesting, we’ll now see how to enter data via the keyboard. The C++ `iostream` library makes this task easy. To obtain input from the keyboard, the `cin` (pronounced “cee-in”) object is used with the extraction operator (`>>`).

If, in Program 2-7, we want to ask our user to enter his speed, days on the road and hours each day, we use the `cin` object to direct the data from the keyboard into the program variables. (Think of the `>>` as pointing the data where it needs to go.) Here is how we use `cin`:

```
//declare the variables we need
double speed, hrsEachDay;           //average rate in mph, and hrs per day
int daysOnRoad;                     //driving days

//now ask the user and use cin to obtain data
cout << "\n Please enter the average speed";
cin >> speed;
cout << "\n Please enter the number of hours you drive each day";
cin >> hrsEachDay;
cout << "\n Please enter the total days on the road.";
cin >> daysOnRoad;
```

When the program runs, it stops when it reaches the `cin` statement. The user must type the data value and hit the Enter key. Once the Enter key is struck, `cin` extracts the value from the input stream (that is, the “pipeline” of data coming from the keyboard) and assigns it to the data variable.

It is possible to enter multiple values with a `cin` statement. The same code could be written so that the user is asked to enter three values at once, like this:

```
cout << "\n Please enter the number of days you're driving,"
<< "\n the speed, and hours driven each day."
<< "\n such as 4 68.0 7.5 ";
cin >> daysOnRoad >> speed >> hrsEachDay;
```

We will spend more time examining how `cin >>` works, and learn its limitations. But for now, consider `cin` as your tool for reading numeric data from the

keyboard. Let's look at another short program that includes reading data from the keyboard, using operators and assignments to gain more practice with these C++ statements. Program 2-8 calculates plumbing hardware parts needed for our garden.

Program 2-8

```
1 //Hardware calculation program.
2 //PVC pipe for your garden.
3 //Declare variables, use operators, obtain user values
4
5 #include <iostream>      //for cout and cin
6 using namespace std;
7
8
9 int main()
10 {
11     int nConnectors, nElbows, nPipes; //variables
12     double pipeLength;
13
14     //Now ask the user to input his data.
15     cout << "\n Welcome to the C++ PVC Plumbing Store";
16
17     cout << "\n\n Please enter the number of ... ";
18     cout << "\n pipe connectors? ";
19     cin >> nConnectors;
20
21     cout << "\n elbows? ";
22     cin >> nElbows;
23
24     cout << "\n Length of pipe in feet? "
25         << " (can be value such as 8.5) ";
26     cin >> pipeLength;
27
28     cout << "\n How many " << pipeLength << " foot pipes? ";
29     cin >> nPipes;
30
31     //We want to calculate total length of pipe.
32     double totalLength;
33     totalLength = pipeLength * nPipes;
34
35     //Calculate total number of plumbing parts.
36     int totalParts = nConnectors + nElbows + nPipes;
37
38     //Write totals to the screen
39     cout << "\n Summary";
40     cout << "\n Number of Connectors: " << nConnectors;
41     cout << "\n Number of Elbows: " << nElbows;
42     cout << "\n Number of " << pipeLength << " foot pipes: " << nPipes;
```



```

43
44     cout << "\n Total Pipe Length (ft): " << totalLength;
45
46     cout << "\n Total Plumbing Parts: "  << totalParts << "\n\n" ;
47
48     return 0;
49 }

```

Program Output

Welcome to the C++ PVC Plumbing Store

```

Please enter the number of ...
pipe connectors? 5
elbows? 3
Length of pipe in feet? (can be value such as 8.5) 5.5
How many 5.5 foot pipes? 9
Summary
Number of Connectors: 5
Number of Elbows: 3
Number of 5.5 foot pipes: 9
Total Pipe Length (ft): 49.5
Total Plumbing Parts: 17

```

Assignment Operator

In C++ the assignment operator (=) takes the value on the right side of the equals sign and places it in the variable on the left side. The following code shows three assignment statements—the numeric value `1534.34` is placed in `balance`, the value of `8` is placed into `nPipes`, and the value of `y` is placed into `x`:

```

balance = 1534.34;
nPipes = 8;
x = y;

```

It is possible to have many assignment operators in one expression. In C++ the assignments begin on the right, and move to the left. In this example, we are placing `0` into `c`, then the value of `c` into `b`, and then `b` into `a`.

```

a = b = c = 0;           //valid assignment setting a, b, c to 0

```

Precedence of Operations

precedence of operations

set of rules that dictates the order in which operations are performed

A program statement may contain many operators. The C++ language includes rules for it to follow in order to execute the statement instructions correctly. ***Precedence of operations*** (or order of operators) in C++ define these rules—meaning it dictates exactly how statements are performed. In layman's terms, this gives us the ability to know which operation is performed first, which operation is performed second, and so on, in a program statement. There must be a convention for the C++ compiler to follow for obtaining a consistent result. Table 2-8 is an abbreviated version of Appendix C and illustrates the precedence for arithmetic and assignment operators.

TABLE 2-8

Arithmetic and Assignment Precedence of Operations

Priority	Operator Type	Operator	Associativity
Highest	Primary	$() \ [] \ . \ ->$	Left to right
	Arithmetic	$*$ $/$ $\%$	Left to right
	Arithmetic	$+$ $-$	Left to right
Lowest	Assignment	$=$	Right to left

Let's look at the temperature conversion example again.

```
F_temp = 9.0/5.0 * C_temp + 32.0; //convert from C to F degrees
```

In this expression, there are four operators: $=$, $/$, $*$, and $+$. The multiplication ($*$) and division ($/$) operators have the highest priority (there are no primary operators in this expression). When this program is executed, these two operations are performed before the others. But which one gets carried out first? **Associativity** tells us. For the arithmetic operators, the associativity is "left to right," meaning that the operator *on the left in the expression* is performed first. So the computer will divide 9.0 by 5.0 and then multiply by the value in the *C_temp* variable. The addition is done next. The assignment operator finishes by placing the result from the calculations into *F_temp*.

The use of a set of parentheses, which are primary operators, means that the code inside the $()$ is performed first. By writing our equation with parentheses, this changes the order of operations.

```
F_temp = 9.0/5.0 *(C_temp + 32.0); //Temp Conversion with ()'s
```

The operations inside the parentheses are executed first, following the precedence of operations. In this statement, the addition is performed first. Next, the division occurs, followed by the multiplication and assignment. Writing the statement in this manner results in an incorrect temperature conversion.

Figure 2-6 illustrates several equations written in C++. Based on the precedence and associativity of the operators, arrows indicate the order in which each calculation is performed.

Data Types and Stored Values When values are assigned into C++ variables, the type of variable dictates what is actually stored in memory. If the programmer enters:

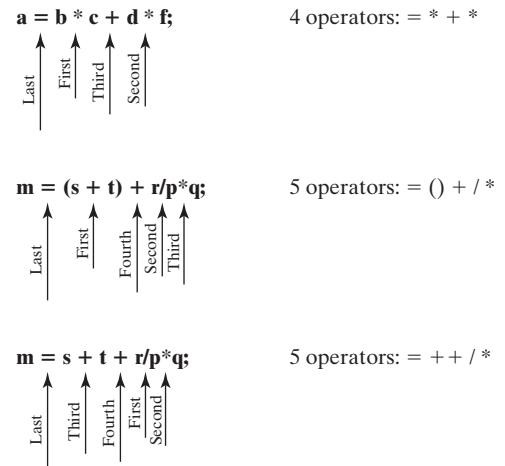
```
double x = 15; // x will actually be stored as 15.00000000000000
```

the double values will hold a value with 14-15 zeros. When you assign a value to an integer, only the whole number portion will be stored. The decimal portion is truncated (*not* rounded). Neither the decimal nor the digits to the right of it are stored.

```
int money = 435.83; // integer money will have 435 stored
```

associativity

specifies the order of operations if operators have the same priority

**Figure 2-6**

Precedence and associativity of operators dictate the order of operations. In each C++ statement the * and / are performed before the + and – operations. The = is performed last.

The compiler will issue a warning if the programmer attempts to assign a value that will be truncated due to the data type. For example, in these two statements the value for pi and nNFLfans is truncated (or changed) due to the limitation of the data type.

```
float pi = 3.141592653589793; // pi will be stored accurately as 3.141593
short int nNFLfans = 56332; // short int limit is 32,767
```

Program 2-9 is a short program that illustrates these concepts. Examine the code, the list of compiler warnings, and the resultant output. Notice how the money value is just 435 (the 83 cents were truncated during the assignment). Note too how the two values of pi (pi and PI) differ in the output. The float pi would only retain the value 3.1415927, the extra incorrect digits are just “trash” that is written when we request 14 digits of decimal precision.

Program 2-9

```
1 //Variable Values Illustration Program
2 //Declare and assign values into variables.
3
4 #include <iostream> //for cout
5 using namespace std;
6
7 int main()
8 {
9
10    int money = 435.83; //will only store 435
11
12    short int nNFLfans = 56332; //attendance at NFL game
13
14    float pi=3.1415926535; //float only holds 7 places
15    double PI = 3.1415926535; //double is good to 15 places
16
```

```

17     double x = 15;           //double has 15 digits of precision
18
19     cout << "\n Values";
20     cout << "\n money (int) " << money;
21     cout << "\n NFL fans (short int) " << nNFLfans;
22
23     //Tell cout to write 14 places of precision
24     cout.setf(ios::showpoint | ios::fixed);
25     cout.precision(14);
26
27     cout << "\n pi as a float " << pi;
28     cout << "\n PI as a double " << PI;
29     cout << "\n x (double) " << x << "\n\n";;
30
31     return 0;
32 }
33

```

Compiler Warnings

Truncation.cpp(12) : warning C4305: 'initializing' : truncation from
 'const int' to 'short'
 Truncation.cpp(12) : warning C4309: 'initializing' : truncation of
 constant value
 Truncation.cpp(14) : warning C4305: 'initializing' : truncation from
 'const double' to 'float'

Program Output

Values
 money (int) 435
 NFL fans (short int) -9204
 pi as a float 3.14159274101257
 PI as a double 3.14159265350000
 x (double) 15.00000000000000

How Does C++ Interpret Constants? When you examine the warnings from Program 2-9, you see two references to constants, a ‘const int’ and a ‘const double.’ It is important for the C++ programmer to understand how C++ interprets constants. A constant value can be either a numeric constant or a textual one. If you are trying to assign a number or text by placing it on the right side of an assignment operator, what data type is it? Sounds confusing, doesn’t it? An example will serve us well.

Figure 2-7 shows the various ways C++ interprets the symbol 5. Written without a decimal point, the 5 is considered an *int*. Written with a decimal point, the 5. or 5.0 is interpreted as a double. In C++, single tick marks ‘5’ makes it a *char*, and *doubl*e quotation marks “5” results in a char array or C-string. Now look at the warning in Program 2-9 and see that C++ is referring to a *const int* (56332) that is being truncated to a *short int*. The 3.1415926535 is a ‘const double’ that is being truncated to a *float*.

C++'s interpretation of constant values

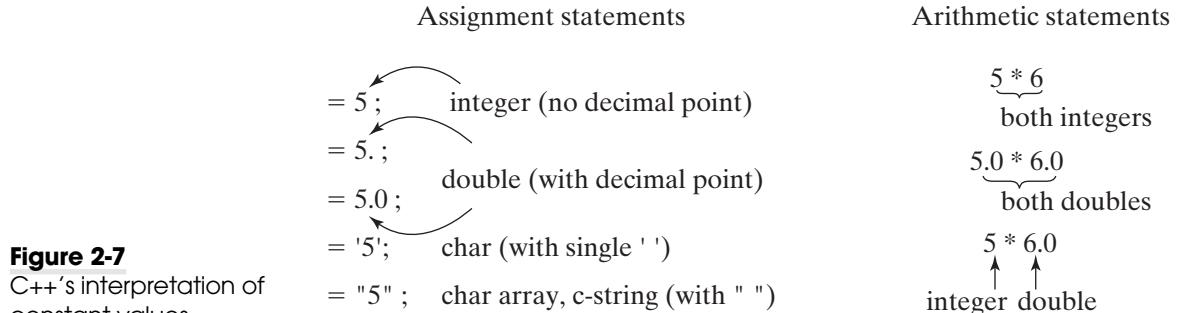


Figure 2-7
C++'s interpretation of constant values.

C++ Does Not Initialize Variables What happens if you write a program and forget to assign values into variables? C++ does not assign any values automatically to your variables. Do not assume values start at zero. The various C++ development environments usually initialize the variables to the largest negative value possible. Usually beginning C++ programmers see the results of no-assignment situation when their programs produce “crazy” output.

In Program 2-10, we present the road-trip program from earlier in the chapter, and notice that we have not assigned any values into the *speed*, *hrsEachDay* or *daysOnRoad* variable. The compiler provides warning messages that variables have been used without having been initialized. (It is very important to always heed the warnings from your compiler!) Examine the code, the warnings and the crazy output!

Program 2-10 Problems occur when you forget to assign values!

```

1 //This program illustrates what happens if the
2 //programmer forgets to initialize his data variables.
3
4 #include <iostream>           //for cout
5 using namespace std;
6
7
8 int main()
9 {
10    //declare our variables here
11
12    double speed, hrsEachDay;      //rate, daily duration
13    int daysOnRoad;               //driving days
14

```

```

15     double dailyMiles, totalMiles, totalHoursDriven;
16
17     //calculate miles driven each day hours x speed
18     dailyMiles = hrsEachDay * speed;
19
20     //calculate total hours by days x hours
21     totalHoursDriven = daysOnRoad * hrsEachDay;
22
23     //total miles is total hours x average rate
24     // (or could have used daily miles x days driven)
25     totalMiles = totalHoursDriven * speed;
26
27     cout << "\n Driving Trip Summary ";
28     cout << "\n " << daysOnRoad << " days on road driving " << speed
29             << " mph for " << hrsEachDay << " hours each day";
30     cout << "\n\n Daily miles: " << dailyMiles;
31     cout << "\n Total driving hours: " << totalHoursDriven;
32     cout << "\n Total miles driven: " << totalMiles << "\n\n";
33
34     return 0;
35 }
```



Warnings

miles.cpp(18) : warning C4700: local variable 'hrsEachDay' used without
 having been initialized
 miles.cpp(18) : warning C4700: local variable 'speed' used without having
 been initialized
 miles.cpp(21) : warning C4700: local variable 'daysOnRoad' used without
 having been initialized

Output

Driving Trip Summary
 -858993460 days on road driving -9.25596e+061 mph for -9.25596e+
 061 hours each day

Daily miles: 8.56729e+123
 Total driving hours: 7.95081e+070
 Total miles driven: -7.35924e+132

lvalue Often programmers see messages concerning *lvalue* and *rvalue*. An *lvalue* is an object that can be on the left side of the assignment operator. An *rvalue* can be on the right side of the assignment operator. C++ expects certain items to be on the left and right sides of the assignment operator. Invalid *lvalue* error messages are given for these expressions in this short program, Program 2-11.

lvalue

an entity that may be found on the left side of an assignment operator

rvalue

an entity that may be found on the right side of an assignment operator

Program 2-11 with compiler errors

```

1 //assign from left to right, not right to left
2
3 #include <iostream>
4 #include <cmath>
5 using namespace std;
6
7 int main()
8 {
9     double x, sqrootX;
10    5.2 = x;           // can't assign from right to left
11
12    sqrt(x) = sqrootX; // can't call sqrt on left of = sign
13
14    cout << "\n Square root of " << x << " is " << sqrootX << "\n\n";
15
16    return 0;
17 }
```

**Compiler Errors**

```

lvalues.cpp(10) : error C2106: '=' : left operand must be l-value
lvalues.cpp(12) : error C2106: '=' : left operand must be l-value
```

Program 2-11

```

1 //assign from left to right, not right to left
2
3 #include <iostream>
4 #include <cmath>
5 using namespace std;
6
7
8 int main()
9 {
10    double x, sqrootX;
11    x = 5.2;           //assign number from right to left
12
13    sqrootX = sqrt(x); //calculate on right, assign to left
14
15    cout << "\n Square root of " << x << " is " << sqrootX << "\n\n";
16
17    return 0;
18 }
```

**Program Output**

Square root of 5.2 is 2.28035

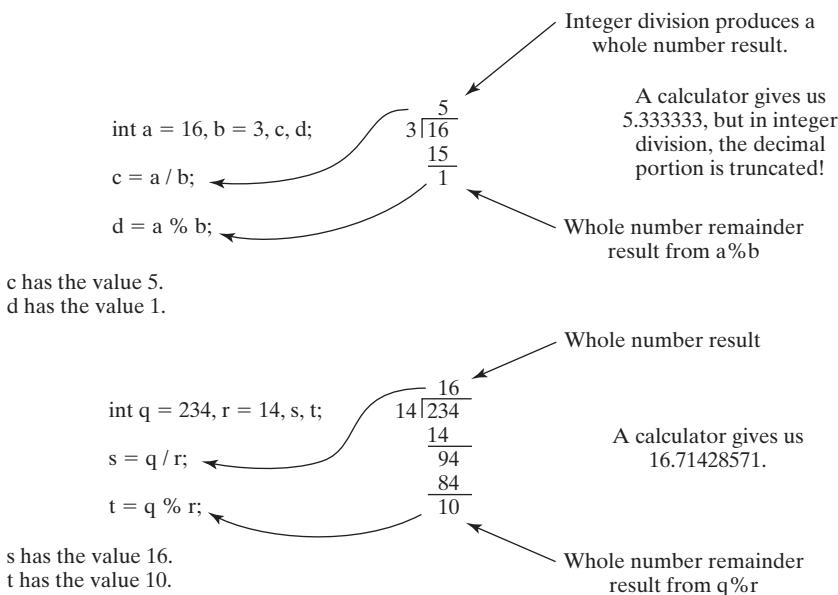
Arithmetic Operators

C++ provides five arithmetic operators: multiplication (*), division (/), addition (+), subtraction (-), and modulus (%). The first four operators are self-explanatory, but the modulus operator is new to many C++ programmers. The **modulus operator** must have integer **operands**, that is, the modulus operator only works on integers, and it returns the whole number remainder in a division. Let's clarify what we mean by operand. If you add two numbers together, such as 5 + 3, the operator is the “+” and 5 and 3 are operands. Figure 2-8 illustrates how both the modulus and division operators work.

Beginning programmers frequently see no use for the modulus operator—however, it is a very convenient operator. For example, if you need to know if an integer is odd or even, the number could be mod’ed with 2 and the result checked for 0 (the number is even) or 1 (the number is odd). This scheme is used to determine if a number is evenly divisible by 10 (or any integer). An integer mod’ed with 10 returns 0 if the number is evenly divisible by 10.

Intermediate Results with Arithmetic Operators C++ has a simple rule for a programmer who is working with the four arithmetic operators. If the two values (operands) on which the operator is working are integers, the result is an integer. If the operands are floats or doubles, the result is a double. If there is one integer and one double or one integer and one float, the result is a double. (Modulus operator works on, and returns, only integers.) Table 2-9 summarizes this rule.

A problem occurs when two integers are divided. The result from this type of division is an integer. It does not matter to what type of variable the division is being assigned. In Program 2-12 we perform division operations using two integers and two doubles. The results of these divisions are assigned into integers and doubles. As the program shows, the integer division result of 16/3 is 5 (decimal portion is truncated) and it does not matter what type of variable receives the result.



modulus operator %

returns the whole number remainder when dividing two integers

operand

the value on which an operator operates

Figure 2-8

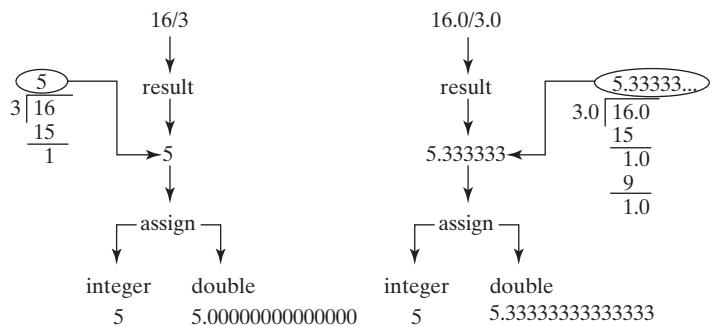
Modulus and division operators

■ TABLE 2-9
Intermediate Results with Arithmetic Operators

Operand Data Types	Result Data Type	Example	Result
Both integer	int	$5 * 4$	20
		$16 / 3$	5
		$7 + 8$	15
		$8 - 2$	6
		$17 \% 5$	2
int float	double	$5 * 4.0$	20.00000000000000
int double			
Both float/double	double	$5.0 * 4.0$	20.00000000000000

Figure 2-9

Intermediate results from integer and floating point (double) division and subsequent assignment results.



When double division occurs and the result is assigned into an integer, the decimal portion is lost, as ints hold only whole numbers. Study the program code, results and Figure 2-9, as together they illustrate this C++ feature.

```

Program 2-12
1 #include <iostream>
2 using namespace std;
3
4
5 int main()
6 {
7     int resInt;
8     double resDouble;
9
10    //First we'll do integer division.
11    //Remember that 16 and 3 are integers in C++.
12    resInt = 16/3;

```

```
13     resDouble = 16/3;
14
15 //Tell cout to write 14 decimal places for our doubles.
16 cout.setf(ios::fixed | ios::showpoint);
17 cout.precision(14);
18
19 cout<< "\n Integer division is 16/3 ";
20 cout << "\n Result assigned to an int: " << resInt;
21 cout << "\n Result assigned to a double: " << resDouble << "\n\n";
22
23 //Second, we'll do the same division but with doubles.
24 //By writing 16.0 and 3.0, C++ regards them as doubles.
25 resInt = 16.0/3.0;
26 resDouble = 16/3.0;
27
28 cout<< "\n Double division is 16.0/3.0 ";
29 cout << "\n Result assigned to an int: " << resInt;
30 cout << "\n Result assigned to a double: " << resDouble << "\n\n";
31
32 return 0;
33 }
```

Output

```
Integer division is 16/3
Result assigned to an int: 5
Result assigned to a double: 5.000000000000000

Double division is 16.0/3.0
Result assigned to an int: 5
Result assigned to a double: 5.33333333333333
```

More Practice with Fractional Calculations Figure 2-10 shows four fractional calculations illustrating operator precedence and integer division and parentheses, being primary operators, can change the results of a calculation.

It is easy to read a series of fractions in an equation and assume that the computer will calculate the final answer in the same manner that we do in an algebra equation or on a calculator. The order of precedence dictates that parentheses expressions are performed first. When the developer is using multiplication and division, the order of operations is from left to right. Take time to learn the precedence of operations and associativity for these operators. Program 2-13 shows the results of our four equations. Can you work through these four examples without peeking at this figure and come up with the same results?

Program 2-13

```
1 //Program to demonstrate operator precedence.
2
```

```

3  #include <iostream>           //for cout
4  #include <iomanip>           //for setw()
5  using namespace std;
6
7  int main()
8  {
9      double X, Y, Z, Q;
10
11     cout << " This program demonstrates how operator precedence "
12         << "\n and () can give different arithmetic results "
13         << "\n with the same numeric values. \n\n" ;
14
15     //Perform four calculations and just vary the ()s.
16     X = 2 + 5.0 * 4 - 6.0 * 7/2;
17     Y = (2 + 5.0) * 4 - (6.0 * 7/2);
18     Z = 2 + (5.0 * 4 - 6.0) * 7/2;
19     Q = 2 + 5.0 * (4 - 6.0 * 7)/2;
20
21     //Write results to 3 places of precision.
22     cout.setf(ios::fixed | ios::showpoint);
23     cout.precision(3);
24
25     //We'll use setw() to have C++ write the values in 8 spaces.
26     //This allows us to line up the output values.
27     cout << "\n X = " << setw(8) << X;
28     cout << "\n Y = " << setw(8) << Y;
29     cout << "\n Z = " << setw(8) << Z;
30     cout << "\n Q = " << setw(8) << Q << "\n\n";
31
32     return 0;
33 }
```

Output

This program demonstrates how operator precedence
and () can give different arithmetic results
with the same numeric values.

X = 1.000
Y = 7.000
Z = 51.000
Q = -93.000



Troubleshooting Using Fractions in Temperature Conversions The intermediate results rule can cause programmers an incredible amount of grief when working with fractional calculations. If a number in an expression is entered without a decimal point, it is treated as an integer. Programmers forget this fact and write code using algebraic-type expressions.

double X $X = 2 + 5.0 * 4 - 6.0 * 7/2;$ $= 2 + 20.0 - 6.0 * 7/2$ $= 2 + 20.0 - 42.0/2$ $= 2 + 20.0 - 21.0$ $= 22.0 - 21.0$ $= 1.0$	double Y $Y = (2 + 5.0) * 4 - (6.0 * 7/2);$ $= 7.0 * 4 - (6.0 * 7/2)$ $= 7.0 * 4 - (42.0/2)$ $= 7.0 * 4 - 21.0$ $= 28.0 - 21.0$ $= 7.0$
double Z $Z = 2 + (5.0 * 4 - 6.0) * 7/2;$ $= 2 + (20.0 - 6.0) * 7/2$ $= 2 + 14.0 * 7/2$ $= 2 + 98.0/2$ $= 2 + 49.0$ $= 51.0$	double Q $Q = 2 + 5.0 * (4 - 6.0 * 7)/2;$ $= 2 + 5.0 * (4 - 42.0)/2$ $= 2 + 5.0 * (-38.0)/2$ $= 2 - 190.0/2$ $= 2 - 95.0$ $= - 93.0$

Figure 2-10

Varying the order of operations can produce different results.

Suppose we need to convert between the Celsius and Fahrenheit temperatures. The conversion equations are:

$$\text{Fahrenheit} = \frac{9}{5}\text{Celsius} + 32$$

$$\text{Celsius} = \frac{5}{9}(\text{Fahrenheit} - 32.0)$$

If the fractional portions of these expressions are written correctly, the programmer has remembered to use 9.0/5.0 and 5.0/9.0, like this (assumes that you have declared the farenheit and celsius variables as doubles):

```
farenheit = 9.0/5.0*celsius + 32.0;
```

and

```
celsius = 5.0/9.0*(farenheit - 32.0);
```

The incorrect way, written as “9/5” and “5/9,” produces integer division results of 1 and 0, respectively, *not* 1.8 and 0.55555, as our calculator gives us.

```
farenheit = 9/5*celsius + 32.0;
```

and

```
celsius = 5/9*(farenheit - 32.0);
```

Program 2-14 demonstrates these two equations coded correctly. Also, we see that the program has the necessary formatting code for the `cout` object so that the program writes the double values with 3 places of decimal precision.



Program 2-14

```
1 //Temperature conversion demonstrating
2 //how to code from C to F and F to C equations.
3
4 //Notice how we end up where we started! :-)
5
6 #include <iostream>      //for cout and cin
7 using namespace std;
8
9 int main()
10 {
11     double celsius, farenheit;
12
13     //Format cout output for 3 decimal places
14     cout.setf(ios::fixed | ios::showpoint);
15     cout.precision(3);
16
17     cout << "\n Welcome to the C++ Temp Converter Program";
18
19     cout << "\n Begin by entering the current temp in F ==> ";
20     cin >> farenheit;
21     cout << "\n You entered " << farenheit << " F"
22         << "\n Let's convert it to Celsius " ;
23
24     //Convert F to C
25     //Notice that we code the fraction correctly!
26     celsius = 5.0/9.0*(farenheit - 32.0);
27     cout << "\n Resultant Celsius temp = " << celsius;
28
29     cout << "\n\n Let's be sure we calculated C correctly"
30         << "\n by converting it back to F ";
31
32     //Convert celsius back to farenheit degrees
33     farenheit = 9.0/5.0*celsius + 32.0;
34     cout << "\n\n Resultant Farenheit temp = " << farenheit;
35
36     cout << "\n Are they the same temps? :-) \n\n" ;
37
38     return 0;
39 }
```

Output

```
Welcome to the C++ Temp Converter Program
Begin by entering the current temp in F ==> 89.5
You entered 89.500 F
Let's convert it to Celsius
Resultant Celsius temp = 31.944
```

Let's be sure we calculated C correctly
by converting it back to F

Resultant Fahrenheit temp = 89.500
Are they the same temps? :-)

When to Include <cmath> The many mathematical functions of C++ include a square root, raising a number to a power, trigonometric functions, and others. Often beginning programmers believe the math library must be included if any math is to be performed in a program. Remember, there are five arithmetic operators that are part of the C++ language—multiply, divide, add, subtract, and modulus. The math library is not needed to perform any of these five operations. If the programmer needs a square root or tangent function for his or her program, the math library must be included in the source code, along with the other preprocessor commands, as shown here.

```
#include <cmath>
using namespace std; // don't forget this!
```

Helpful Hints Frequently, algebraic expressions need to be written in C++. C++ code looks similar to algebra. C++ has parentheses () and dot operators, but they have different meanings in C++ than in algebra. Examples shown in Table 2-10 illustrates the correct way to code algebraic equations. Also break up complicated equations into two or three steps because it is easier to debug discrete steps rather than a massive equation.

Increment and Decrement Operators

The increment (++) and decrement (--) operators are useful because they provide a quick way to add or subtract one (1) from a variable. These operators are used typically with integers. When you write a line of code like this:

```
++i; or i++;
```

it is equivalent to

```
i = i + 1;
```

Beginning C++ programmers often ask about the placement of the operator and if there is a difference between the following two statements:

```
++i; //prefix operator ++ is before the variable
i++; //postfix operator ++ is after the variable
```

The answer is, there is no difference. When the increment or decrement operator is used with a variable, as shown above, there is no difference—however, there is a difference when the operators are used in an assignment statement.

TABLE 2-10

Algebraic Equations and C++ Expressions

Algebra	C++: The Right Way	C++: The Wrong Way
$v = (a + b)(c - d)$	<code>double v, a, b, c, d;</code> <code>v = (a + b) * (c - d);</code>	<code>double v, a, b, c, d;</code> <code>v = (a + b)(c - d); //Note 1</code>
$SA = \pi \cdot \text{radius}^2$	<code>double SA, pi, rad;</code> <code>SA = pi * rad * rad;</code> <code>//OR See Note 2.</code> <code>SA = pi * pow(rad, 2);</code>	<code>double SA, pi, rad;</code> <code>SA = (pi)(rad)(rad); //Note 1</code> <code>//OR</code> <code>SA = pi * rad **2; // Note 3</code> <code>SA = pi * rad ^2; // See Note 3</code>
$a = \frac{c + b}{x - y}$	<code>double a, b, c, x, y;</code> <code>a = (c+b)/(x-y);</code>	<code>double a, b, c, x, y;</code> <code>a = c + b/x - y; //Note 4</code>
$m = \frac{\sqrt{x \cdot 3y}}{w}$	<code>double x, y, w, m;</code> <code>m = sqrt(x * 3*y)/w; //Note 2</code>	<code>double x, y, w, m;</code> <code>m = sqrt(x.3y)/w; //Note 5</code>
$f(x) = \frac{2}{3} \sin(x - 0.3)$	<code>double x, fofx;</code> <code>fofx=2.0/3.0*sin(x-0.3);</code>	<code>double x, f(x);</code> <code>f(x)=2/3sin(x-0.3); //Note 6</code>

Note 1: To perform multiplication, the `*` operator must be used. The `()()` does not mean multiplication in C++.

Note 2: The `cmath` library needs to be included to use the square root and power functions.

Note 3: The `rad**2` or `rad ^2` are not valid ways to do exponentiation in C/C++.

Note 4: Division has higher priority; `b/x` would be done first.

Note 5: The dot operator `(.)` is not multiplication in C++.

Note 6: Cannot name variables `f(x)`.

The prefix operator will increment/decrement and then assign; whereas the postfix operator will assign and then increment/decrement. See Table 2-11.

Accumulation Operators

The accumulation operators (`+=`, `-=`, `*=`, and `/=`) provide quick ways to write assignment expressions when it is necessary to accumulate values. For example, there are two ways to add a value to a sum variable:

```
sum = sum + x;           //one way to write
sum += x;                //using the += accumulation operator
```

Beginning programming students sometimes find these operators a bit obscure, but with time, programmers prefer them when coding statements that calculate averages

TABLE 2-11

Pre-Fix and Post-Fix Increment and Decrement Operators

Operator	Job	Format	Equivalent To	Start i = 5, then ...
Prefix increment	Add 1 to <i>i</i> then assign <i>i</i> into <i>m</i> .	<i>m</i> = <i>++i</i> ; <i>m</i> = <i>i</i> ;	<i>i</i> = <i>i</i> + 1; <i>i</i> = <i>i</i> ;	<i>i</i> = 6 <i>m</i> = 6
Postfix increment	Assign <i>i</i> into <i>m</i> and then add 1 to <i>i</i> .	<i>m</i> = <i>i++</i> ;	<i>m</i> = <i>i</i> ; <i>i</i> = <i>i</i> + 1;	<i>m</i> = 5 <i>i</i> = 6
Prefix decrement	Subtract 1 from <i>i</i> and then assign <i>i</i> into <i>m</i> .	<i>m</i> = <i>--i</i> ;	<i>i</i> = <i>i</i> - 1; <i>m</i> = <i>i</i> ;	<i>i</i> = 4 <i>m</i> = 4
Postfix decrement	Assign <i>i</i> into <i>m</i> and then subtract 1 from <i>i</i> .	<i>m</i> = <i>i--</i> ;	<i>m</i> = <i>i</i> ; <i>i</i> = <i>i</i> - 1;	<i>m</i> = 5 <i>i</i> = 4

or other total values. The sum operator is the commonly used, and we'll see examples in later chapters. The other accumulation operators are:

```
diff = diff - x;           //subtract a value, or
diff -= x;                // use the -= accumulation operator
product = product * x;    //multiply a value, or
product *= x;              // use the *= accumulation operator
quot = quot/x;            //divide a value, or
quot/= x;                 //use the /= accumulation operator
```

2.7

Miscellaneous Topics: #define, const, and casting

#define

The `#define` is a preprocessor directive (as is `#include`) that gives the compiler instructions. Usually the statements are located at the top of the file. The `#define` statement is a symbolic constant and allows the compiler to perform a straight substitution in the code when it is compiled. The form of the `#define` statement is:

```
#define symbolic_name character_sequence
```

Note that there is no semicolon at the end of the statement. An example of a `#define` is:

```
#define PI 3.14159265
```

The `#define` statements are not actual C++ statements that perform an instruction but are compiler substitutions and are performed when the code is compiled. The `#define` is convenient for coding numeric constants and gives the programmer an easy way to remember words instead of numbers. If the programmer needs to have numeric constants, the software is written with `#define` statements as shown in Program 2-15.

#define

preprocessor
directive, compiler
symbolic substitution

**Program 2-15**

```
1 //Program illustrating the use of #defines
2
3 #include <iostream>
4 using namespace std;
5
6 #define PI 3.14159265
7 #define RADIUS 5.257
8
9 int main()
10 {
11     double circleArea;
12
13     //The area of a circle is PI * Radius * Radius
14
15     circleArea = PI * RADIUS * RADIUS;
16
17     cout << "\n Circle Summary ";
18     cout << "\n The circle's radius is " << RADIUS;
19     cout << "\n The value of PI is " << PI;
20     cout << "\n The circle's area is " << circleArea << "\n";
21
22     return 0;
23 }
```

Output

```
Circle Summary
The circle's radius is 5.257
The value of PI is 3.14159
The circle's area is 86.8212
```

When the compiler reads and builds the object code for this program, it substitutes the character sequence from the `#define` statement directly into the code where the `symbolic_name` appears, except inside “” of literal strings. When the compiler sees the symbolic names `PI` and `RADIUS` in Program 2-15 source code, it simply substitutes the symbols from the `#define` statement. We still see `PI` and `RADIUS` in the source code, but if we could read the machine code, the `main` function would look like this:

```
circleArea = 3.14159265* 5.257 * 5.257;
cout << "\n Circle Summary ";
cout << "\n The circle's radius is " << 5.257;
cout << "\n The value of PI is " << 3.14159265;
```

Be careful that you do not accidentally type a semi-colon in a `#define` statement, as the semi-colon will be substituted into the statements as well. If your `#define` statements were:

```
#define PI 3.14159265;
#define RADIUS 5.257;
```

Your equation would be interpreted like this:

```
circleArea = 3.14159265; * 5.257; * 5.257;;
```

The compiler is confused with the semi-colon followed by the asterisk (`;`*) and believes you are attempting to use an indirection operator. We'll learn about the indirection operator (*) in Chapter 5. The compiler generating these error messages when compiled:

```
Defines.cpp(15) : error C2100: illegal indirection
Defines.cpp(15) : error C2100: illegal indirection.
```

Common C++ conventions are to use capital letters for `#define` constants and to place them at the top of the file so they are easy to locate. These `#define` statements are used for defining important numbers or identifying data files in a program. It also gives you to change one line of code rather than every time the value is used. Here are three more examples:

```
#define MAX 100
#define FILE_IN      "F:\data\input.dat"
#define HOURLY_RATE 8.50
```

The `const` Modifier

C++ has an access modifier, `const`, that is used in variable declaration statements. When used in a declaration statement, the constant's variable initial value remains the same throughout the program. For the most part, you cannot change the value of a variable that has been declared with the `const` modifier—it remains constant. The form is:

```
const data_type variable_name = initial_value;
```

An example is:

```
const double x = 7.1111;
```

The value in the `const` variable must be assigned when declared and the program then is not allowed to change this value. The following short program illustrates how different compilers implement this rule:

Program 2-16 with Compiler Error

```
1 //Attempt to changing a const variable COMPILE ERROR
2
3 #include <iostream>
4 using namespace std;
5
6 int main ()
7 {
```

const

modifier specifying that the variable is to remain a constant value

```

8      const double x = 7.1111; //declare x to be a constant value
9      x = 6.2;      //now try to change it
10
11      cout << "\n X = " << x;
12      return 0;
13  }

```

Compilers always catch this error, and the message takes on two forms (depending on the compiler). Microsoft compilers state that the left-hand value is a *const* object (and hence not changeable by the program):

```
ChangeConst.cpp(9) : error C2166: l-value specifies const object
```

whereas the UNIX/Linux based compilers refer to the constant as a read-only variable:

```
constvar.cc:9: assignment of read-only variable 'x'
```

Are *consts* Better Than *#define*?

The *const* modifier and *#define* statement both provide the programmer with a method for fixing a value and using it throughout the entire program. But there is one fundamental difference! The *const* modifier specifies that a variable's value may not be changed. The *#define* statement is merely a symbolic substitution that the compiler performs behind the scenes. The *#define* value is not associated with a program variable.

When you declare a variable in a C++ program, you are able to see that variable and its value in the debugger. If you *#define* a value, you cannot see the value in the debugger. Great discussions can ensue around the question of whether or not the use of *#define* is good style. Because *const* variables can be seen in the debugger, their values are not a mystery. On the other hand, *#defines* are easy ways to set values in one location, and they do not need to be passed between functions. What you do is up to you.

Data Casts

C++ has certain rules and ways that it handles various data types and the operations we perform with them. The language interprets the value 2 as an integer, and if written with a decimal point (2. and 2.0) as a double. If we assign a double into a float, we encounter a truncation warning. That is, if we coded this statement,

```
float q = 2.3456789;
```

C++ interprets the 2.3456789 value as a double, and you are warned that you are converting a “*const double* to a *float*” and there is a possible loss of data. Even if you code this line:

```
float r = 2.0;
```

you'd still be warned of a “possible loss of data”, as 2.0 is still considered a double data type.

Another problem is the integer division feature of C++. Let's say you keep track of your Spider Solitaire² games that you play in one year, and you write a C++ program that calculates your average number of games per day and win percentage over a year's time. Here are a few lines for your game stats program: Can you spot a problem?

```
int daysInYear = 365;           //assume 2007, non-leapyear
int gamesPlayed = 1693;         //wow, that's a lot of games (?wins?)
int gamesWon = 115;            //not very good, are you? ;)
float aveGamesPerDay = gamesPlayed/daysInYear;
float winPercent = gamesWon/gamesPlayed * 100;
```

In both of these calculations, the program is performing integer division and the intermediate results give us incorrect values. (As coded, our results would be 4 games per day, and 0% win rate.)

How do we fix this program? We need to still maintain our integer values—after all, we do not have partial games played or won, and obtain accurate game results? Also, how do we assign constant values into data types so that we do not receive truncation warnings? The answer is to perform a ***data cast***. Data casting is an operation where one type of data is transformed into another type of data.

C++ Preferred Data Type Cast The concept of casting a variable (or *data cast*) is an operation in which the value of one type of data is transformed into another type of data. This means that if we have an integer, we can tell C++ to change it to a float or double. C++ has four types of casting expressions and the `static_cast` is the one most commonly required in C++ programs. Refer to the C++ Keywords list in Table 2-4 and the Keyword Dictionary in Appendix B for more information on these casting expressions. For now, let's concentrate on the one cast that we'll be using.

The general form of a static cast is:

```
static_cast<data_type>(value)
```

where `data_type` is the new type of data for the value. Here is how we convert a constant double into a float so we can assign it into a float type variable. This statement is telling C++ to convert the double value of 2.0 into a float value.

```
float r = static_cast<float>(2.0);
```

In this next statement, we cast a character variable that contains the letter 'B' into an integer variable named `number`.

```
char letter = 'B';
int number = static_cast<int>(letter);
```

If we wish to calculate accurately our Spider daily game average and win percentage, we need to avoid performing integer division. As long as one of the division values is a float (or a double), we obtain a floating point result. Program 2-17 shows the correct way to cast our integers and obtain correct results.

data cast

an operation in which the value of one type of data is transformed into another type of data

²Spider Solitaire is a popular two-deck solitaire game that is played with one, two, or four suits.

Program 2-17

```
1 //Use static_cast to compute Spider game stats correctly.
2
3 #include <iostream>           //for cout
4 #include <iomanip>           //for setw()
5 using namespace std;
6
7 int main()
8 {
9     int daysInYear = 365;      //assume 2007, non-leapyear
10    int gamesPlayed = 1693;    //wow, how many games did you win?
11    int gamesWon = 115;       //not very good, are you? ;)
12
13    float aveGamesPerDay, winPercent;
14
15    //cast the integer number of games played into a float
16    aveGamesPerDay= static_cast<float>(gamesPlayed)/daysInYear;
17
18    //cast the games played into a float
19    winPercent = static_cast<float>(gamesWon)/gamesPlayed * 100;
20
21    //write results with 2 decimal places of accuracy
22    cout.setf(ios::fixed | ios::showpoint);
23    cout.precision(2);
24
25    cout << "\n Spider Game Statistics for One Year \n\n"
26        << setw(6) << gamesPlayed << " Games Played \n"
27        << setw(6) << gamesWon << " Games Won \n"
28        << setw(6) << aveGamesPerDay << " Ave Games Per Day \n"
29        << setw(6) << winPercent << " % Win Rate " << "\n\n";
30
31    return 0;
32 }
```

**Output**

Spider Game Statistics for One Year

1693 Games Played
115 Games Won
4.64 Ave Games Per Day
6.79 % Win Rate

Old-Style Casting Methods The C language provided several ways to perform data casting operations, and these techniques are still supported by the current C++ language. The new C++ programmer should use the *static_cast* expression to ensure that her code is compatible with future editions of the C++ language. It is possible that you will see source code that uses these other casting styles, so for completeness, they are presented here.

One technique of data casting was performed using parentheses enclosing the new (desired) data type. For example, our assignment and calculations code could be written like this:

```
float r = (float)(2.0);           //use (float) before the 2.0
float r = 2.0F;                  //place the letter F or f after the 2.0
char letter = 'B';
int number = (int)letter;
float aveGamesPerDay = (float)gamesPlayed/daysInYear;
float winPercent = (float)gamesWon/gamesPlayed * 100;
```

2.8

More Details on Keyboard Input and Screen Output

Our programs in Chapter 2 have illustrated the use of the *cin* and *cout* objects, and how to do some formatting of the output. This section presents more information on these two objects, with tables to reference for exact spelling and usage. In the Practice section of this chapter you will see more formatting examples.

Escape Sequences

As a general rule, C++ takes the stream of data that you write in the *cout* statement and displays what it receives. The *cout* object uses double quotes to indicate the start of the output text and then again to find the end of the text. In this statement:

```
cout << "Hello World";
```

the double quotes (" ") show the start and end of the text to be printed.

When the backslash \ is combined with certain characters inside the text, the *cout* object knows that the \ indicates that there may be something special with the character that follows it. For example, when we write:

```
cout << "Hello World \nHow are you today?";
```

the "How are you today?" is written on the next line. The \n tells cout to write the following text on the next line. The \ symbol tells C++ "to escape from" the normal interpretation and to do some thing special when writing output. The \n is known as an *escape sequence*. There are several escape sequences and they are listed in Table 2-12.

Here's a question for you. How do you write a C++ statement so that you see the \ in the output? Or how do you see "" quotation marks in the output? The escape sequences are required in the cout statements so that C++ knows it is supposed to write the symbols, not use the symbols for outputting direction. In order to see the title of a book enclosed in quotation marks, or actually have a \ in the output, your *cout* statements need the appropriate escape sequence. See Program 2-18.

escape sequence

a \ and a character that has special meaning in formatting output. For example \n means write on new line.

■ TABLE 2-12
Escape Sequences in C++

Escape Sequence	Purpose
\a	Beep
\b	Backspace
\f	Formfeed
\n	Newline
\r	Return
\\\	Backslash
\"	Single quote
\'	Double quote
\xdd	Hexadecimal notation
\ddd	Octal notation

Program 2-18

```

1 //An example showing escape sequences
2 //to write \ and " in formatted output.
3
4 #include <iostream>           //for cout
5 using namespace std;
6
7 int main()
8 {
9     cout << "\n How are you liking "
10    << " \"C++ Programming Today 2nd Edition\" ? ";
11
12    cout << "\n The \\n is an escape sequence for a newline.\n\n";
13
14    return 0;
15 }
```

Output

How are you liking "C++ Programming Today 2nd Edition" ?
The \n is an escape sequence for a newline.

ios Formatting Flags

ios formatting flags
on-off switches that you set to control formatting aspects of your output

The **ios formatting flags** can be thought of as on-off switches that control how output data is written. If you want your data values written to 4 decimal places of accuracy, and right-justified within a certain amount of space, you need to specify turn on/off the correct switch (or set the flag). Table 2-13 shows the list of *ios* flags that are used with *cout*. Program 2-19 illustrates how the precision and fixed flag work together to produce a variety of output formats.

TABLE 2-13

ios Formatting Flags

ios Flag	Purpose
<i>skipws</i>	Leading whitespace characters are discarded.
<i>left</i>	Output is left justified.
<i>right</i>	Output is right justified.
<i>internal</i>	A numeric value is padded to fill a field.
<i>dec</i>	Write output in decimal.
<i>hex</i>	Write numeric output in hexadecimal.
<i>oct</i>	Write numeric output in octal.
<i>showbase</i>	Shows the base of numeric values (such as 0x for hex).
<i>showpoint</i>	Shows a decimal point and trailing zeros for all floating point values.
<i>uppercase</i>	Shows scientific and hex notation in uppercase, such as 3.4E3 or 0XF2.
<i>showpos</i>	Shows leading plus sign.
<i>scientific</i>	Floating point data is displayed in scientific notation, such as 3.4e4.
<i>fixed</i> ^a	Floating point values are displayed in normal notation.
<i>unitbuf</i>	I/O system is flushed after each output operation.

^aDefault shows six decimal places. If neither fixed nor scientific is set, compiler chooses the appropriate method.

Program 2-19

```

1 // Practice with iosflags
2 // We'll use the endl to flush the cout buffer.
3
4 #include <iostream>           //for cout
5 #include <iomanip>          //for endl
6
7 using namespace std;
8
9 int main()
10 {
11
12     double pi = 3.141592653589793;
13     double feet = 5280;
14     double number = 123.456789;
15
16     //first let's just write the values
17     cout << "\n First write: not setting any flags"
18         << "\n Pi = " << pi
19         << "\n Feet = " << feet
20         << "\n Number " << number << endl;
21
22     //if we just do precision, it fixes how many
23     //spaces are used to write values
24     cout.precision(5);

```

```

25
26     //showpoint says to write the decimal point
27     //cout remembers that precision is set to 5
28     cout.setf(ios::showpoint);
29     cout<< "\n Second write: set prec to 5 and showpoint"
30             << "\n Pi = " << pi
31             << "\n Feet = " << feet
32             << "\n Number = " << number << endl;
33
34     //if you set fixed and precision, it sets decimal places
35     cout.setf(ios::fixed);
36     cout<< "\n Third write: set fixed with precision of 5"
37             << "\n Pi = " << pi
38             << "\n Feet = " << feet
39             << "\n Number = " << number << endl;
40
41     return 0;
42 }
```

Output

First write: not setting any flags

Pi = 3.14159

Feet = 5280

Number 123.457

Second write: set prec to 5 and showpoint

Pi = 3.1416

Feet = 5280.0

Number = 123.46

Third write: set fixed with precision of 5

Pi = 3.14159

Feet = 5280.00000

Number = 123.45679

Stream Iomanipulators

C++ provides you with the ability to place manipulators directly in the output stream. You need to include the `<iomanip>` library to use them. One handy manipulator is the `setw()`, which sets the field width for the variable output that follows. The following line of code uses a `setw()` manipulator to reserve five spaces in which to write the value of the variable `x`:

```
cout << "\n The value of x is " << setw(5) << x;
```

setw()

manipulator function
that is placed in the
output stream, sets
the field width of the
variable that follows it

Justification, either left or right, is handy to use when you are trying to align your output. You need to use the `setw()` in conjunction with a justification flag so that C++ has the field width as well as the justification specifications. Program 2-20 shows how you can place the flags in the cout statement itself to control the

appearance of your output. Remember that once you set a flag, or a precision value, it remains until you change it. Also, for most of the formatting flags, you can set it outside of the cout statement, as we did in the previous program, or within the statements as we do here.

Program 2-20

```
1 // Practice using iomanipulator and iosflags
2 // We use the setw and scientific notation flag here.
3
4 #include <iostream>           //for cout
5 #include <iomanip>           //for endl
6
7 using namespace std;
8
9 int main()
10 {
11
12     double pi = 3.141592653589793;
13     double feet = 5280;
14     double number = 123.456789;
15
16     //The precision can be placed in the stream
17
18     cout<< "\n First write: set precision to 5"
19         << setprecision(5)
20         << "\n      Pi = " << pi
21         << "\n      Feet = " << feet
22         << "\n      Number = " << number << endl;
23
24     //fixed and precision, we get 5 decimal digits
25     //cout still remembers that precision is set to 5
26     cout<< "\n Second write: set fixed with precision of 5"
27         << setiosflags(ios::fixed)
28         << "\n      Pi = " << pi
29         << "\n      Feet = " << feet
30         << "\n      Number = " << number << endl;
31
32     //Data is automatically right-justified within
33     //the setw() field width size
34     cout<< "\n Third write: use setw(12)"
35         << "\n      Pi = " << setw(12) << pi
36         << "\n      Feet = " << setw(12) << feet
37         << "\n      Number = " << setw(12) << number << endl;
38
39     //Now write the data in scientific notation.
40     //Need to unset the fixed flag to see sci notation.
41     cout.unsetf(ios::fixed);
42     cout<< "\n Fourth write: scientific notation"
43         << setiosflags(ios::scientific)
```

```

44          << "\n      Pi = " << setw(12) << pi
45          << "\n      Feet = " << setw(12) << feet
46          << "\n      Number = " << setw(12) << number << endl;
47
48      return 0;
49  }

```

Output

```

First write: set precision to 5
Pi = 3.1416
Feet = 5280
Number = 123.46
Second write: set fixed with precision of 5
Pi = 3.14159
Feet = 5280.00000
Number = 123.45679
Third write: use setw(12)
Pi = 3.14159
Feet = 5280.00000
Number = 123.45679
Fourth write: scientific notation
Pi = 3.14159e+000
Feet = 5.28000e+003
Number = 1.23457e+002

```

2.9 Get Started Using Classes and Objects, the C++ string

Recall from Chapter 1 that in object-oriented programs, we design our program components so that we have classes and use objects. You can think of a class as a job description—which lists the tasks that must be performed by an individual doing that job. The object is the actual individual who is performing that job. The C++ language provides many classes for us to use in our programs. In fact, we have been using the *cout* and *cin* objects in this chapter. When you include *<iostream>* library, C++ provides you the pre-defined objects, *cout* and *cin*. What we mean by pre-defined object is that they are already set up and ready to go! The *cout* object knows how to write to the console window. It is performing the job defined in the *ostream* class. The *cin* object knows how to read data from the keyboard. It is performing the jobs (tasks) defined in the *istream* class.

C++ has a very useful class that we will begin using now. The C++ ***string class*** is contained in the *<string>* library and is used to handle text data in your program. The *string* class provides the programmer with many operators and other functions for searching, assigning, and manipulating data within the *string* object. Appendix G “Partial C++ Class Reference” provides details and you should refer to it for more information. Now let’s see how to use a *string* object.

string class

a C++ class that provides the tools for working with textual data

Program 2-21 illustrates how to declare string objects and use them in a program. Here we declare three objects, assigning our favorite language into the object named *favLang*. We ask for and read the programmer's name and school into the program. We use the *getline* function to read the string information. The *cin* object only reads to the first space, and stops, so we don't want to read string data with anything but *getline()*.

Program 2-21

```
1 //C++ string objects
2
3 #include <iostream>           //for cout, cin, getline
4 #include <string>            //for string
5
6 using namespace std;
7
8 int main()
9 {
10    //declare 3 string objects, initialize one of them
11    string name;
12    string school;
13    string favLang = "C++, of course";
14
15    cout << "\n C++ Programming Student Demographics ";
16
17    //ask the user to enter her name and school
18    //use getline function to read from cin into string
19    cout << "\n Please enter your name: ";
20    getline(cin, name);
21
22    cout << "Please enter your school: ";
23    getline(cin, school);
24
25    cout << "\n Programming Student Data:"
26        << "\n Name: " << name
27        << "\n School: " << school
28        << "\n Favorite language: " << favLang << endl;
29
30    return 0;
31 }
```

Output

```
C++ Programming Student Demographics
Please enter your name: Hannah White
Please enter your school: Central New Mexico Community College
Programming Student Data:
Name: Hannah White
School: Central New Mexico Community College
Favorite language: C++, of course
```

Each object in your program comes equipped and ready to perform the tasks that are defined in its class (job description). The way we ask an object to do one of its tasks is by using the object's name, the dot operator (which is a period) and the name of the task. For example, if we have two string objects, we can ask each one to tell us how many characters they each have in their data. Look at these statements and see that we are asking each object to tell us how many characters are in its data—that is, we are asking the object to tell us the size of its data.

```
string baseballTeam = "Albuquerque Isotopes";
string teamMascot = "Orbit";
int teamSize = baseballTeam.size();           //teamSize is assigned 20
int mascotSize = teamMascot.size();           //mascotSize is assigned 5
```

The following short program asks the user to enter a sentence and a word. We find the size of our two strings, and then search the sentence for the word. Actually the string class has a pre-defined task named “find.” When we use the object name, the dot operator and “find” we can pass into the find function what we want it to look for in its own data. We are giving that object an order—go and find this string in your data. (Look at Figure 2-11. It shows how Visual C++ 2005 gives you

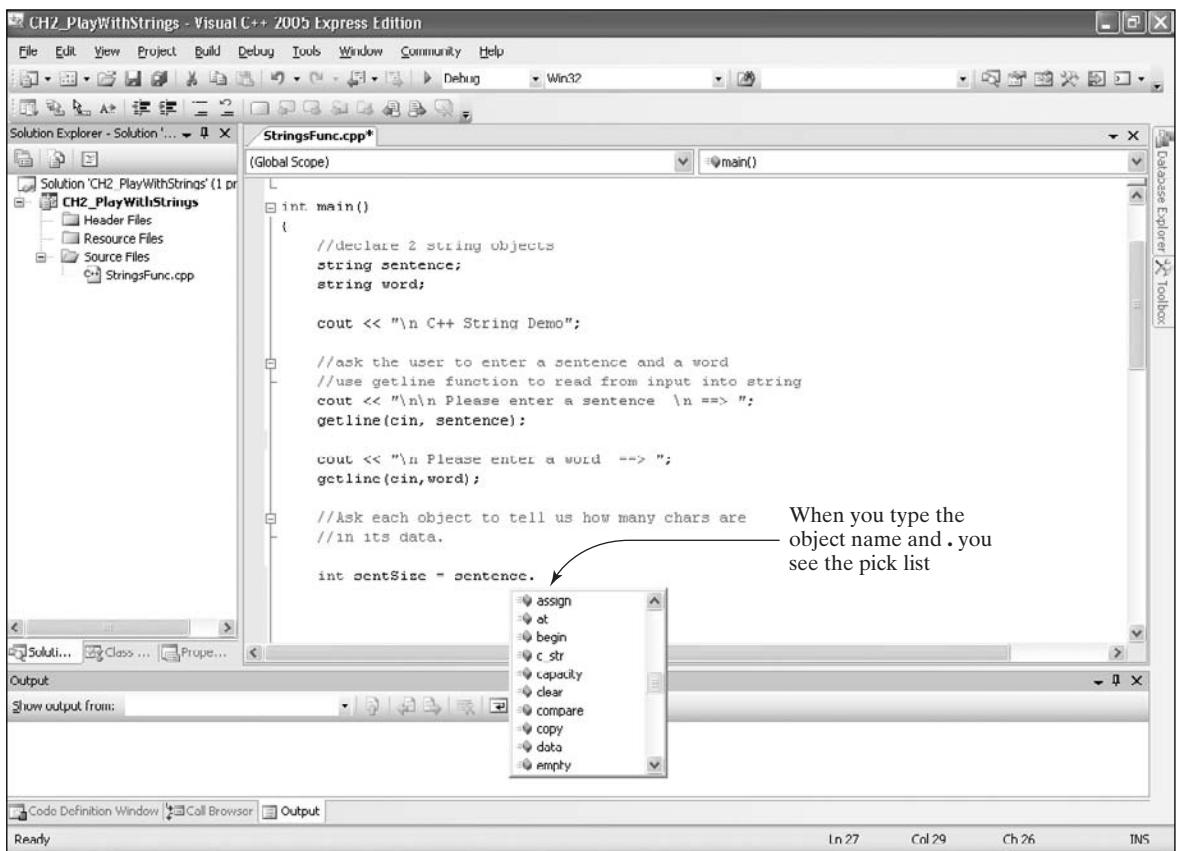


Figure 2-11

Sentence is a string object in our program. When I use the object name sentence (and dot operator) you will see the functions associated with that class.

the class' pick list (of functions) when I use the dot operator with the sentence string in Program 2-22.) If the string is in the object's data, it returns the location where that string begins. (C++ is zero-indexed, meaning that it counts the first character as 0.) The *find* function in the string class is built so that it returns a -1 if it does not find the string in its data.

It is important for you to see the big picture in how we're declaring objects and then using the object to perform tasks. After all, this is what object-oriented programming is all about!

Program 2-22

```
1 //C++ string objects, size and searching
2
3 #include <iostream>           //for cout, cin, getline
4 #include <string>            //for string
5
6 using namespace std;
7
8 int main()
9 {
10    //declare 2 string objects
11    string sentence;
12    string word;
13
14    cout << "\n C++ String Demo";
15
16    //ask the user to enter a sentence and a word
17    cout << "\n\n Please enter a sentence  \n ==> ";
18    getline(cin, sentence);
19
20    cout << "\n Please enter a word  ==> ";
21    getline(cin,word);
22
23    //Ask each object to tell us how many chars are
24    //in its data.
25
26    int sentSize = sentence.size();
27    int wordSize = word.size();
28
29    //Search to find where the word is in the sentence.
30    //Ask the sentence object to find the word???
31    int wordInSentence = sentence.find(word);
32
33    cout << "\n String Demo Results"
34        << "\n Sentence: " << sentence
35        << "\n Word: " << word
36        << "\n Sentence size " << sentSize
37        << "\n Word size " << wordSize
```

```
38      << "\n Word location in Sentence: "
39      << "(-1 not in sentence ): " << wordInSentence << endl;
40
41      return 0;
42  }
```

Output

C++ String Demo

```
Please enter a sentence
==> C++ is my favorite programming language.
```

```
Please enter a word ==> favorite
String Demo Results
Sentence: C++ is my favorite programming language.
Word: favorite
Sentence size 40
Word size 8
Word location in Sentence: (-1 not in sentence ): 10
```

2.10 Practice!

Time to Learn the Debugger!

Before we jump into our first set of practice programs, now is an excellent time to learn how to use the powerful debugging tool in Visual C++ 2005 Express! Turn back to the Appendix I, “Microsoft Visual C++ 2005 Express Edition Debugger” and read through the first few examples. This appendix guides you through debugging steps for several programs, including our first practice problem below. The sooner you learn to use the debugger, the happier you’ll be!

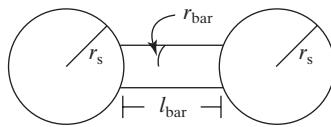
Calculate the Volume of a Dumbbell

In our first practice program, we are going to calculate the volume of a dumbbell that consists of two spheres, each being attached to each end of a cylindrical bar. Often times programming students are able to “do the math” on paper, but have trouble translating the math into a C++ program. Figure 2-12 shows a two-step approach to writing this program. Notice how the pencil and paper calculations on the left-side of the figure makes the transition to C++ straightforward. We draw a picture and label the various mathematical parts. Once we see the solution, we write the steps into C++ code.

The Program 2-23 requirements include asking the user for pertinent data needed for the dumbbell. The output should show the input data and the resultant values to three decimal places. Be sure that the output data is aligned neatly.

First:

determine variables, inputs
and required math first



r_s = radius sphere

r_{bar} = radius bar

l_{bar} = length bar

vol = sphere + cylinder + sphere

Math

$$= 2 \times \left(\frac{4}{3}\pi r_s^3\right) + \pi r_{\text{bar}}^2 \cdot l_{\text{bar}}$$

Test case :

$$r_s = 5.0$$

$$r_{\text{bar}} = 1.0$$

$$r_{\text{length}} = 12.0$$

$$= 2 \times \left(\frac{4}{3}\pi(5)^3\right) + \pi(1)^2 \cdot 12$$

$$= 2 \times (523.59877) + 37.69911$$

$$= 1084.89666$$

Second:

write

into
C++
program

```
int main()
{
    // variables
    double rs, rbar, lbar;
    double sphVol, barVol, totalVol;
    // ask for dumbbell dimension

    // calc sphere volume

    // dumbbell vol = 2 * sphere + bar

    // write results
}
```

Figure 2-12

Use a Two-step approach for writing a program to calculate the volume of a dumbbell.

Program 2-23

```
1 //Calculate volume of a dumbbell.
2
3 #include <iostream>           //needed for cout and cin
4 #include <iomanip>           //needed for setw()
5 #include <cmath>               //for power function
6
7 using namespace std;
8
9 int main()
10 {
11     float pi = static_cast<float>(3.14159265);
12     float rs, rbar, lbar;
13     float sphVol, barVol, totalVol;
14
15     //obtain dumbbell dimensions
16     cout << "\nPlease enter radius of the end-spheres: ";
```

```

17     cin >>rs;
18
19     cout << "\nPlease enter radius and length of the bar: ";
20     cin >>rbar >>lbar;
21
22     //calculate sphere vol
23     sphVol = 4.0/3.0*pi*pow(rs,3);
24
25     //calculate the bar volume
26     barVol = pi*pow(rbar,2)*lbar;
27
28     //total volume is bar volume + 2(sphere volumes)
29     totalVol = barVol + 2.0*sphVol;
30
31     //write results
32     cout.setf(ios::fixed | ios::showpoint); //write w/ 4 digits of prec
33     cout.precision(3);
34
35     cout << "\n Dumbbell Volume Results \n";
36     cout << setw(15) << "Sphere Radius" << setw(15) << "Bar Radius" <<
37         setw(15) << "Bar Length" << setw(15) << "Total Volume" << endl;
38
39     cout << setw(15) << rs << setw(15) << rbar <<
40         setw(15) << lbar << setw(15) << totalVol << endl;
41
42     return 0;
43 }
```

Output

```

Please enter radius of the end-spheres: 5.0
Please enter radius and length of the bar: 1.0 12.0
Dumbbell Volume Results
Sphere Radius  Bar Radius  Bar Length  Total Volume
      5.000        1.000       12.000      1084.897
```

Character Data As Decimal, Hex, and Octal and *ios* Formatting

C++ uses the ASCII character code set to represent character data. The ASCII code set is in Appendix D. Program data is stored in binary format (1s and 0s) that can then be interpreted as data in decimal (base 10), octal (base 8), or hex (base 16). Appendix E covers these various numeric formats. Program 2-24 asks the user to enter two different characters, then shows these characters in their decimal, hex, and octal forms. We use the *ios* formatting flags to tell *cout* how to write the numeric data. Refer to Appendix D to double-check the program results shown here.

Program 2-24

```
1 //Casting Chars into Ints,
2 //Using ios flags to view in octal and hex notation
3
4 #include <iostream>           //for cout and cin
5 #include <iomanip>          //for setw
6
7 using namespace std;
8
9 int main()
10 {
11     char char1, char2;
12     int nChar1, nChar2;
13
14     cout << "\n Enter two characters ==> ";
15     cin >> char1 >> char2;
16
17     cout << "\n\n Your Characters: " << setw(5) << char1
18         << setw(5) << char2;
19
20     nChar1 = static_cast<int>(char1);    //cast into integers
21     nChar2 = static_cast<int>(char2);
22
23     cout << "\n      Decimal form: " << setw(5) << nChar1
24         << setw(5) << nChar2;
25
26     //turn off decimal flag, turn on octal flag
27     cout.unsetf(ios::dec);
28     cout.setf(ios::oct);
29     cout << "\n      Octal form: " << setw(5) << nChar1
30         << setw(5) << nChar2;
31
32     //turn off octal, turn on hex
33     cout.unsetf(ios::oct);
34     cout.setf(ios::hex);
35     cout << "\n      Hex form: " << setw(5) << nChar1
36         << setw(5) << nChar2 << endl;
37
38     return 0;
39 }
```

**Output**

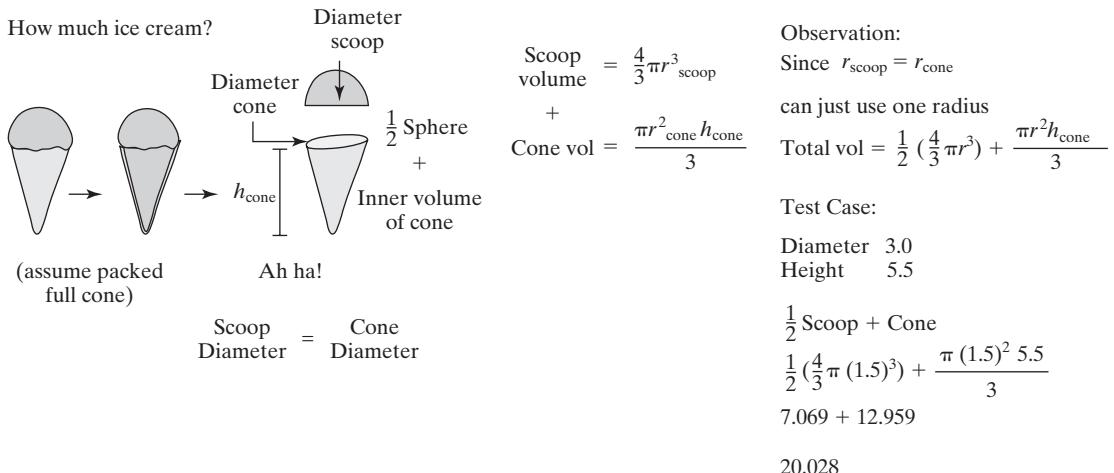
Enter two characters ==> % B

Your Characters: % B

Decimal form: 37 66

Octal form: 45 102

Hex form: 25 42

**Figure 2-13**

Calculate total ice cream in a one-scoop cone.

I Scream, You Scream, We All Scream for Ice Cream!

What is your favorite flavor of ice cream? Chocolate mint? Strawberry? Vanilla with toffee chips? Such are the difficult choices we face in life. Calculating the volume of a single dip ice cream cone gives the new C++ programmer an interesting problem to solve. To calculate the total volume of ice cream (we'll assume the cone is packed full) the programmer must view the solution as a sum of two parts: the top half of the scoop, and the volume of the cone. Figure 2-13 illustrates this. In Program 2-25, we follow the layout shown in the figure. Isn't programming fun? I wonder if there is any ice cream in the freezer. After I write this program, I'll go check.

Program 2-25

```

1 //Ice cream volume calculation program
2 #include <iostream> //for cout, cin, getline
3 #include <cmath> //for power
4 #include <string> //for string object
5 using namespace std;
6
7 //Practice using a #define to set PI's value
8 #define PI 3.14159265
9
10 int main()
11 {
12     //Note: the ice cream cone and scoop have the same diameter.
13     double diameter, coneHeight;
14
15     string flavor;
```

```
16      cout << "\n Welcome to the C++ Single Scoop Ice Cream Parlor";
17
18      cout << "\n Please enter your desired flavor: ";
19      getline(cin,flavor);
20
21      cout << "\n Please enter the cone height (inches): ";
22      cin >> coneHeight;
23
24      cout << "\n Please enter the cone diameter (inches): ";
25      cin >> diameter;
26
27
28 //Now calculate volume of ice cream
29 //First calculate the cone volume
30 double coneVol, radius;
31
32 radius = diameter/2.0;
33 coneVol = (PI * pow(radius,2) * coneHeight)/3.0;
34
35 //next calculate the volume of the scoop
36 double scoopVol = 4.0/3.0 * PI * pow(radius,3);
37
38 //total volume is entire cone plus 1/2 the scoop
39 double totalIceCream = coneVol + scoopVol/2.0;
40
41 //write output
42 cout.precision(1);
43 cout.setf(ios::fixed | ios::showpoint);
44
45 cout << "\n Results \n Your desired flavor is " << flavor;
46 cout << "\n Cone size " << diameter << " by " << coneHeight;
47 cout.precision(3);
48 cout << "\n Total volume of " << flavor <<
49     " ice cream is " << totalIceCream << " cubic inches " << endl;
50
51     return 0;
52 }
```

Output

```
Welcome to the C++ Single Scoop Ice Cream Parlor
Please enter your desired flavor: Chocolate Mint
Please enter the cone height (inches): 5.5
Please enter the cone diameter (inches): 3
Results
Your desired flavor is Chocolate Mint
Cone size 3.0 by 5.5
Total volume of Chocolate Mint ice cream is 20.028 cubic inches
```

Water Facts and Stock Tanks for Livestock

Water. We all know this tasteless, odorless substance is necessary for all forms of life. But did you know that a gallon of water weighs 8.32867 pounds? Or that there are 231 cubic inches of water in a gallon? Or that one cubic foot of water contains 7.48 gallons? If you have the time and inclination you can derive these values yourself, or look them up on the Internet. These water facts are fertile ground for developing interesting problems for a C++ student.

For this last practice problem, let's investigate the capacity of a stock tank—that is, how many gallons of water it contains. Stock tanks are large metal containers used to provide drinking water for livestock. In Program 2-26 we'll ask our cowboy for the diameter and height of his stock tank and how full he wishes his tank to be (50%, 75%, 90%, etc.) Our program will calculate the number of gallons needed to fill the tank to this level. We'll make use of the water facts above to solve this problem.

Figure 2-14 shows our approach to determining tank capacity. The circular stock tank is just a cylinder. If we calculate the volume of the cylinder in cubic inches, we can convert to gallons by dividing this volume by 231. (Recall that 231 cubic inches are found in one gallon of water.) This gallon value is for a full tank. If our cowboy wants the tank three-fourth full, we multiply the value by 75%.

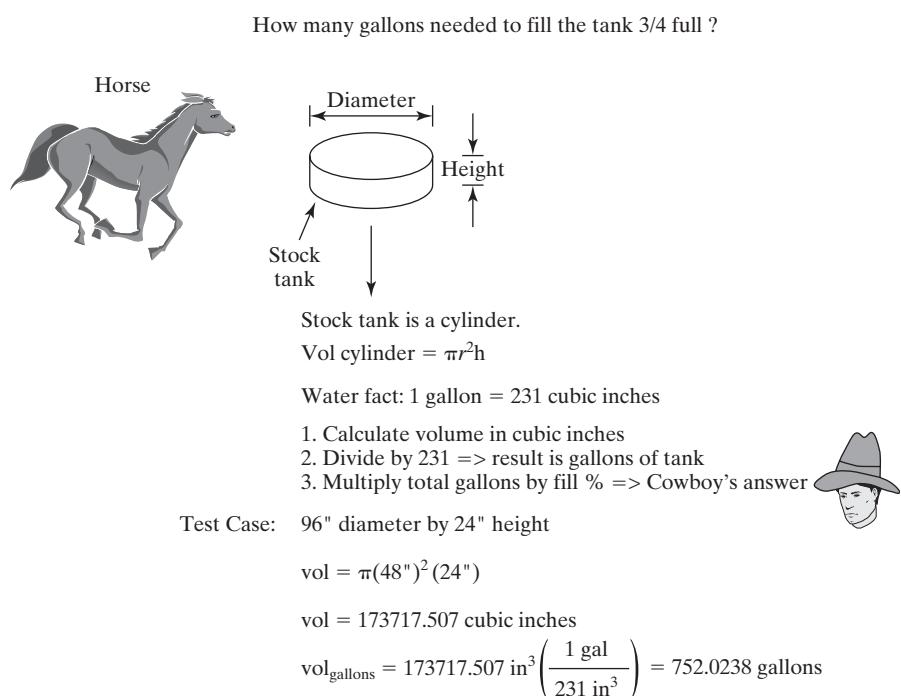


Figure 2-14

Algorithm for the stock tank calculator program.

Program 2-26

```
1 //Program to calculate the number of gallons in
2 //a cowboy's stock tank
3
4 #include <iostream>           //for cout, cin, getline
5 #include <cmath>              //for power
6 #include <string>             //for string object
7
8 using namespace std;
9
10#define PI 3.14159265
11
12int main()
13{
14    //declare variables
15    double tankDia, tankHeight;
16    double fillAmount;
17
18    cout << "\n      Hi Cowboy! Welcome to the "
19          << "\n C++ Circular Stock Tank Capacity Calculator";
20
21    cout << "\n Please enter your tank's diameter in inches: ";
22    cin >> tankDia;
23
24    cout << "\n Please enter the tank's height in inches: ";
25    cin >> tankHeight;
26
27    cout << "\n Please enter fill percentage"
28          << "\n 100 for 100%, \n 50 for 50%, \n 25 for 25%, etc. ";
29    cin >> fillAmount;
30
31    //Now calculate volume of the tank
32    //Use the cylinder formula vol = pi*r*r*h
33    double tankVol, tankRadius;
34
35    tankRadius = tankDia/2.0;
36    tankVol = PI * pow(tankRadius,2) * tankHeight;
37
38
39    //next calculate convert to gallons
40    double tankGallons = tankVol/231.0;
41
42    double desiredFillGal = tankGallons * (fillAmount)/100.0;
43    //write output
44    cout.precision(1);
45    cout.setf(ios::fixed | ios::showpoint);
46
47    cout << "\n Results "
```

```
48             << "\n Stock tank capacity = " << tankGallons << " gallons";
49
50     cout << "\n Filled to " << fillAmount << " % " << " requires "
51             << desiredFillGal << " gallons " << endl;
52
53     return 0;
54 }
```

Output

```
Hi Cowboy! Welcome to the
C++ Circular Stock Tank Capacity Calculator
Please enter your tank's diameter in inches: 96
Please enter the tank's height in inches: 24
Please enter fill percentage
100 for 100%,
50 for 50%,
25 for 25%, etc. 90
```

Results

```
Stock tank capacity = 752.0 gallons
Filled to 90.0 % requires 676.8 gallons
```

REVIEW QUESTIONS AND PROBLEMS**Short Answer**

1. What is the job of the compiler? What does the linker do? Describe the differences between a compile error and a link error.
2. Name five keywords in C++.
3. Is “main” a keyword in C++? Explain your answer.
4. Why does a C++ program need to have a *main* function?
5. What is the difference between a data type and a variable?
6. What is meant by precedence of operations?
7. What is the purpose of a *#include* statement?
8. If you have a double value in your program and just set the precision for the cout statement to 4, what do you see? What do you see if you set the precision and the *ios::fixed* flag?
9. How can the *#define* statement be used to establish the value of PI in your program? Explain where the *#define* statement should be located.
10. When you assign the value 5.628 into a *float* variable, how can you avoid getting the data truncation warning? Why do you get a warning?
11. Where is an escape sequence used?

12. Why is the `getline` function the preferred way to read string values into your program instead of the `cin` object?
13. What happens if you use the modulus operator with two floating point values?
14. What library contains the `setw()` manipulator?
15. Why is it necessary to place “using namespace std;” in your program? Where should it be placed?
16. How is it possible for your program to obtain the data you type in from the keyboard? Explain.
17. If you divide an integer by an integer and place the result into a double, is the decimal portion of the division maintained? Explain your answer.
18. What is the purpose of a project in a C++ development environment?
19. What are the two types of comment styles in C++?
20. How would a programmer ensure that addition occurred before multiplication in his program?
21. If you convert a fractional value into its decimal equivalent (i.e., $\frac{1}{2}$ to 0.5), how can you be sure to obtain an accurate result? Explain your rationale.
22. Which is the best C++ comment style to use?
23. Is it true that if you do not assign values into your variables, then C++ automatically assigns zero into them? Explain.
24. Look up the functions in the C++ string class, and list three of them. Describe their purpose.
25. What is the range of values for a 4-byte integer?
26. Identify valid variable names for the following, and state why the invalid names are not allowed:

default	ten#s	convert-2-pds	_header_h	9_lives
CalcVol	*stars*	pens&pencils	definePI	triangle_area

27. Identify valid variable names for the following, and state why the invalid names are not allowed:

main	+value	DogsAndCats	Bank \$note	_red_ball
Bird brain	\$_for_nothing	1_4_U	U_4_1	Sleeping Dogs
28. Select the appropriate data type and variable (strings included) for each of the following program variables:

Program Data	Data type	Variable name
Area of a rectangle		
Air pressure in a jet’s tire		
Number of passengers		
Student’s name		
Blood chemistry value		
\$ earned in a month		

Program Data	Data type	Variable name
Eyeglasses prescription (diopters)		
Home Address		
Number of pets		
Miles per hour		

29. Select appropriate data type and variable (strings included) for each of the following program variables:

Program Data	Data type	Variable name
Pounds per square inch		
Volume of a cubic box		
Mother's full name		
Parking rate per hour in \$		
Number of cats in a litter		
Price of a gallon of milk		
Credits for a degree		
Town of birth		
Hourly wage in \$		
Total taxes due		

Debugging Problems

For the source code examples in Problems 30 to 34, identify the compiler errors, and state what is needed to eliminate the error(s).

30.

```
#Include <iostream>
int main
{
    float x;
    y = +8.0;
    cout << x << and << y values;
}
```

31.

```
// This is a little program
that calculates a modulus answer.
int main()
{
    double x,y,answer;
    answer = x%y;
    cout << "\n\n answer = << answer;
    return 0;
}
```

32.

```
// Read in the user's name from the keyboard
and show it to the screen. */
#include <string>
```

```
int main()
{
    string Name;
    cout "Please enter your full name.      ";
    cin >> Name;
    cout << "\n\n your name is " << answer;
    return 0;
}
```

33.

```
#include<iostream>
using namespace standard;
int main()
{
    int a,4_for_Fun;
    cout << Enter a;
    cin >> a;
    4_for_Fun = a;
    return 0;
}
```

34.

```
int Main()
{
    int a,b,c;
    int a = 5;
    float $_per_month, price;
    cout << "Enter pay per month and price.";
    cin >> price >> $_per_month;
    return 0;
}
```

Reading the Code

What values will be in the boxes (these represent the computer's memory) once the lines of code in Problems 35 through 39 have been performed? Be sure to show decimal points and full precision if the variable type is capable of holding that data!

35. (There are two compiler warnings with this code. Can you spot them?)

```
int main()
{
    float x = 4.12345678901230, z = 2;
    int a = 6, b = 6.882;
    double r = 3.12345678901234567;
```

x	z	a	b	r

36. (There are two compiler warnings with this code. Can you spot them?)

```
int main()
{
    float x = 234.12345678901230, z = 22.8;
    int a = 44417, b = -0.333 , c = -5;
    double r = 3.12345678901234567;
    long e = 99999;
    char f = '+';
```

x	z	a	b
c	r	e	f

37. (There are three compiler warnings with this code. Can you spot them?)

```
int main()
{
    float a = 4.0, b = 8.0, c = 1.5;
    int x = 5, y = 7.5, z = 19.0;
    float q, r;
    int s, t;
    s = x + z/y*c;
    t = b/a * b*x + c;
    q = y * a+a * c;
    r = z % x+b/a;
```

a	b	c	x	y
z	q	r	s	t

38.

```
int main()
{
    int x = 76, y = 12, z = 5, c = 3;
    double q, r;
    int s, t;
    s = x/c + y/z + c;
    t = x/z + y*c/z;
    q = x/c + y/z + c;
    r = t*z + s*c/z;
```

x	y	z	c
q	r	s	t

39.

```
int main()
{
    int x = 7, y = 2, z = 45;
    double d1 = 7.0, d2 = 2.0;
    int n1, n2;
    double r, s,t;
    n1 = z/x;
    n2 = d1/d2;
    r = z/y;
    s = d1/d2 + x/y*z;
    t = d2/x + z/d1;
```

x	y	z	d1	d2
n1	n2	r	s	t

Programming Problems

Write a complete C++ program. At the start of your program use a `cout` statement(s) to write your name and program title to the screen.

- 40.** Determine the number of cans of paint needed to paint one room of a house (walls and ceiling). You need to ask the user to enter the room's width and length and ceiling height (floor to ceiling distance). Also ask your user to give you an estimate (in %) of wall space that consists of doors and windows in the room. (You do not paint doors or windows.) Figure the total wall space. The estimated door/window area will then be subtracted from it. Don't forget to add the ceiling area to obtain the total painted area. You'll need to ask the user how many square feet a gallon of paint covers. Report the total square footage of painted area, exact amount of paint needed, and whole gallons of paint needed. (You'll round up exact paint to the next whole gallon.) Report the dimensions of the room, too. Write your results to two decimal places.
- 41.** Write a complete C++ program that demonstrates the difference between integer division and the modulus operation. Write a program objective to the screen. Ask the user to enter two integer values. Perform the division and modulus operations on these two numbers and assign the results into integers. Next, perform the division and modulus operations on user's two integer and assign the results into float variables. Write the two values and four operation results to the screen. Use four decimal places of precision for the floating point values. Include descriptive comments.
- 42.** The area of a regular octagon is

$$\text{Area-Octagon} = 4.828a^2$$

where a is the length of one side. Write a complete C++ program that asks the user to enter the size of the octagon (side). Calculate and print the area to three decimal places of accuracy. Use a `#define` statement for the multiplicative constant (4.828) and the `pow` (see `cmath` library) function to find the side-squared value.

- 43.** The volume of a square-based pyramid is:

$$Vol - Pyramid = \frac{A * h}{3}$$

where A is the area of the base and h is the height. Write a complete C++ program that asks the user to enter the necessary information about the pyramid, calculate the volume and print the results (to two decimal places) as well as all dimensional information.

- 44.** The surface area and volume of a sphere is defined to be:

$$SA - Sphere = 4\pi r^2$$

$$Vol - Sphere = \frac{4}{3}\pi r^3$$

where r is the radius of the sphere. Write a program that asks the user to enter the name of two different types of spheres, such as a golf ball and a basketball. Next, ask for the radii of both balls. Report the volume and surface of each ball in your program output.

- 45.** The C++ Traffic Supply Company produces traffic safety cones (orange and yellow) commonly seen around roadway construction sites. The cones have a rectangular base and are hollow so you can stack them for easy transport and storage. We'll assume a closed tip. This program will calculate the amount of molten plastic required to make one cone. (Draw a picture to help visualize your traffic cone!)

The base of the C++ traffic cone cone is 16" square. The cone tip is 28" above the ground. The inside diameter opening at the base of the cone is 12". In order to produce these traffic cones, our company pours melted plastic into molds producing the cone. The cone has a 1" thickness along the cone and the base. Write a C++ program that calculates the cubic inch (and gallons) of plastic required to produce one traffic cone. Hint: you'll need to determine the volumes of the outer cone and inner cone and subtract. When you calculate the base, find the volume for the solid base and subtract the volume for the center hole. Report the dimensions and intermediate results as well as final volume value. Remember that there are 231 cubic inches in one gallon. Your program should write out pertinent dimensions as well as the volumetric results.

- 46.** Write a program that acts as a mortgage payment calculator. It asks the user to enter the amount of the loan (principle), the annual interest rate, and the number of years of the loan. It then reports the monthly payment. You will find this

equation useful:

$$M = \frac{Pi}{\left[q \left(1 - \left[1 + \left(\frac{i}{q} \right) \right]^{-nq} \right) \right]}$$

M = monthly payment n = number of years

P = principle of the loan (amount borrowed)

i = interest rates where 5% is 0.05 q = number of payments per year

Report the principle, number of years, monthly payment, and total interest paid. Be sure to report the dollar values showing two digits of decimal accuracy.

Work through the algebra on paper using two sets of test cases (\$100,000 at 5% for 30 years and \$50,000 at 6.4% for 15 years). Search on the web and find two different Mortgage Calculators. Plug in these same numbers and verify that the web-based calculators give you the same monthly payment value as your pencil and paper results. If these answers all match, then you know you have two good test cases to verify that your program works correctly! (Why am I asking you to find two web-based calculators, not just one?)

47. The C++ Fly A Kite Company builds a variety of kites, including the classic Diamond Flier for kids. These kites have vertical and horizontal poles, which cross at right angles. The company requires customers to order the same size kites. To ensure survival from kite-crashes, the four outer edges of the kite are reinforced with thick wire. The nylon fabric is wrapped around the wire and stitched into place. In order to accurately fit the kite material, the fabric is measured and cut as if each pole was one inch longer on the ends. Each kite requires that the wire be 10% longer than the actual kite perimeter so that the ends are twisted and secured. We'll write a program to help the kite builder determine the material for an order of kites. (Draw a picture!)

The program asks the kite builder to enter the name of the order, the desired color scheme, the number of kites, and the lengths of the cross-bars of a kite (vertical and horizontal). You'll need to ask how far the shorter, horizontal bar is positioned on the longer-vertical one. These dimensions requested in inches, are for the actual size of the finished kite! Your program should then report the order name, color scheme and number of kites. Report the total area of fabric required to build one kite (includes fold over area for stitching), wire, and total pole length for each kite. Also report total fabric, poles and wire for the entire order. Report fabric in square yards and poles and wire in feet and inches. These equations will help. The Pythagorean theorem is very useful for geometric problems. It is:

$$c^2 = a^2 + b^2$$

where c is the hypotenuse of a right triangle and a and b are the sides. Remember that the area of a right triangle is

$$\text{Area Right Triangle} = \frac{1}{2} \text{base} * \text{height}$$

48. The C++ Fly A Kite Company also builds the classic Delta kite, which is triangular shaped. The line attaches to a center flap of fabric sewn into the center axis of the kite. Two poles are sewn into the outer edges providing structural support. These kites are very stable and fly well in turbulent winds. The larger the kite, the stronger the required line because the kite generates lift as it flies! (If you aren't careful, you can be towed behind this type of kite!) For now, write a program that asks the user to enter the desired kite color scheme and length of the wide-bottom edge of the kite. Also ask for the length of the center pole (center of base-to-tip length). Search the web for Kites, you'll see many! Calculate the actual surface area of the kite and the required length of the wing-side poles. Report the color scheme, dimensions, and area of the kite. (Draw yourself a picture of the kite to help you visualize the different sections of this triangular problem! Pythagoras is your friend!)
49. Examine Appendix D, "ASCII Character Codes" and note the various characters found at the high end of the sequence. (See Tables D-1 and D-2.) Write a program that contains 15–20 lines of `cout` statements that produce an image of some sort of face. The face should be identifiable, such as the face of a man or dog, etc.
50. Calculate the number of gallons of water in a rectangular pond. Ask the user to enter the depth, length, and width of the pond (all dimensions are in feet, and use `float` variables). Report the pond dimensions and total gallons. Remember that 231 cubic inches of water is equivalent to one gallon.
51. A 55 gallon drum is cylindrically shaped container. (Typically a drum has an inside diameter of 24 to 36 inches.) Write a C++ program that asks the user the inside diameter of his drum, then calculate and show the resultant inside height. This equation might help you.

$$\text{Vol_Cylinder} = \pi * \text{Radius}^2 * \text{Height}$$

52. The C++ Water and Ice Company needs to determine the volume of purified water it uses during May through September. The company produces 10 pound bags of ice. Twice a week the truck loads the bags and makes deliveries to grocery stores in the area. Each store has a freezer where the bags of ice are stored. A full freezer holds 16 stacks of ice bags; each stack is 20 bags. (That is, each freezer holds a maximum of 320 bags of ice.) During each delivery, the C++ company fills the freezer to 80% capacity. On average, for each freezer, the delivery person has to add 50% of the freezer capacity. Write a C++ program that asks the user to enter the number of grocery stores in the area. Calculate the number and weight of the total bags of ice produced for each delivery during these 5 months and the total bags and volume of purified water required the entire of five months. A gallon of water weighs 8.346 pounds. You may assume these are 22 weeks in May through September.
53. The C++ Water and Ice Company also produces liter and half-liter bottles of purified water. The bottles are packaged in 24-item cases. The delivery person makes stops at local warehouse stores (such as Costco, Sam's Club, etc.) and delivers pallets of these drink cases. Each pallet holds 80 cases of water. Write a C++ program that asks the warehouse owner to enter the number of full and

half-liter pallets to be delivered by our company. Your program should report the total number of cases of each drink, total servings as well as total liters, gallons, and weight of the product. One gallon of water is equivalent to 3,785 cubic centimeters (or 3.785 liters).

- 54.** The C++ Golf Resort and Spa C++ is located in the New Mexico high desert, and has several man-made lakes on the golf course. During the summer, the lakes drops a few inches every week due to evaporation. Occassionally there is an afternoon thunderstorm, but generally the lakes must be refilled via water pumped from a nearby reservior. The water hazard on the 18th hole covers 2.5 acres. The groundskeeper needs to keep the lake level constant. Write a program to help him with this task. The program should ask him the number of inches the lake has dropped for the week and then report the number of gallons required to refill it. An acre of covers 43,560 square feet.



3

Control Statements and Loops

KEY TERMS AND CONCEPTS

binary operator
branch
conditional statement
evaluate a condition
jump statement
logical operator
loop
loop altering statement
loop index
nested statements
operand
relational operator
same as operator
ternary operator
unary operator

KEYWORDS AND OPERATORS

break
case
continue
default
do
else
for
if
switch
while
logical operators (`&&` / `||` / `!`)
relational operators
 (`>` `=` `<` `=<`)
relational operators (`==` / `!=`)
conditional operator (`? :`)

CHAPTER OBJECTIVES

- Present the concept of program control and logic statements.
- Illustrate relational and logical operators in C++.
- Demonstrate conditional branching statements using *if* and *switch* statements.
- Compare the similarities between *if* and *switch* statements.
- Demonstrate jump statements.
- Describe the three methods of loop controllers: the *for*, *while*, and *do while* statements.
- Show examples of good coding style.
- Show five common mistakes made when learning to code control statements.
- Demonstrate how the proper use of code indentation makes code easy to read and debug.
- Present common compiler errors.
- Introduce the C++ vector class and demonstrate how it is used in C++ programs.



Decisions, Decisions!

Any computer programming language must provide the tools for checking conditions or values of variables, and depending on the condition or value, make decisions as to which part of the program to execute. What do we mean, check variable conditions? Often in programs we need to be able to determine if one number is greater than another, or if two numbers or strings of text are the same, or count the number of times we've done something. For example, if we try to open a file that contains program data there must be a way to test: "File, are you open?" We need to have some way to control the program "flow." Suppose we allow the user to select from a list of options for the program, how do we check what options are selected and make the program do what the user wishes?

C++ provides the programmer a standard set of comparison operators. These operators can check two values and return a true or false answer. The language has control statements so that certain parts of the code are executed or not. This chapter presents the nuts and bolts of how to write these comparisons and control statements, as well as many C++ program examples. Sit tight, keep reading and practice writing programs using these statements. The material here needs to be second nature to the C++ programmer.

3.1

Relational and Logical Operators

In C++ relational and logical operators are used to evaluate conditional statements. A **conditional statement** is used to determine the state (or states) of a variable (or variables). The result of the evaluation is a 1 (true) or a 0 (false). **Relational operators** are shown in Table 3-1; **logical operators** are shown in Table 3-2. Relational and two logical operators (AND, **&&**, and OR, **||**) are binary; a **binary operator** expects two values or **operands**. The NOT operator (**!**) is a unary operator; a **unary operator** requires only a single operand. See Figure 3-1.

conditional statement

statement in which a condition is evaluated and the result determines which path the program control takes

relational operator

an operator that evaluates **>**, **>=**, **<**, **<=**, **==**, and **!=** conditions

logical operator

an operator that evaluates AND, OR, or NOT conditions

binary operator

operator that requires two operands

operands

C++ operators work on operands—in the expression **a + b**, **a** and **b** are operands and **+** is the operator

unary operator

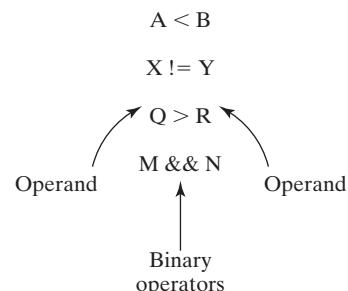
an operator requiring only one operand, such as an increment operator, **+** or **--**

■ TABLE 3-1
Relational Operators in C++

Relational Operator	Tests For	Example
>	greater than	A > B Is A greater than B?
>=	greater than or equal to	A >= B Is A greater than or equal to B?
<	less than	A < B Is A less than B?
<=	less than or equal to	A <= B Is A less than or equal to B?
==	same as	A == B Is A the same as B?
!=	not the same as	A != B Is A not the same as B?

■ TABLE 3-2
Logical Operators in C++

Logical Operators	Tests For	Example
&&	AND	A && B Return 1 if both A and B are 1; otherwise return 0.
	OR	A B Return 1 if either A or B is 1.
!	NOT	!A If A is 1, it is changed to 0. If A is 0, it is changed to 1.



Binary operators require two operands.

The NOT and increment operators are unary and requires one operand.

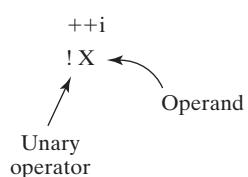


Figure 3-1
Binary and unary operator examples in C++.

Unary operators require one operand.

■ TABLE 3-3

Examples of Relational and Logical Statements

Testing For	Example
Is A greater than B and C greater than D?	(A > B && C > D)
Is E greater than H or E greater than D?	(E > H E > D)
Is A the same as M or the same as Q?	(A == M A == Q)
Is B greater than C and C less than E?	(B > C && C < E)

Evaluating Expressions and Precedence of Operators

It is possible to write conditional statements using both relational and logical operators. Table 3-3 illustrates several combinations of these types of statements. When a programmer writes relational and logical statements in C++ (as in arithmetic operations), the precedence and associativity of operators dictate the order of the operations. Table 3-4 presents the precedence and associativity for several operators. (See also Appendix C, “Operators in C++.”) The arithmetic operator precedence is higher than that for relational operators. The relational operator precedence is higher than that for logical operators. These facts are important when writing conditional statements.

When relational and logical operators are used in an expression, the end result is either a 1 (true) or a 0 (false). Figure 3-2 presents two examples using both relational and logical operators. It shows the manner in which C++ comes to a resultant value. Remember that C++ evaluates only one operator at a time.

3.2

if Statements

C++ provides a flexible *if* statement structure that allows the programmer to build almost any series of conditional statements. The *if* statement is based on a relational and/or logical expression that evaluates to true or false. Before we jump into

■ TABLE 3-4

Arithmetic, Relational, and Logical Precedence of Operations

Priority	Operator Type	Operator(s)	Associativity
Highest	Primary	() [] . ->	Left to right
	Unary	++ -- & * !	Right to left
	Arithmetic	* / %	Left to right
	Arithmetic	+ -	Left to right
	Relational	< <= > >=	Left to right
	Relational	== !=	Left to right
	Logical	&&	Left to right
	Logical		Left to right
	Assignment	=	Right to left

Example 1

$5 + 8 < 14 - 2 \parallel 6 > 3$

The + and - have highest precedence.
The + goes first (it's on the left).

$13 < 14 - 2 \parallel 6 > 3$

Now the -

$13 < 12 \parallel 6 > 3$

Next the <

$0 \parallel 6 > 3$

Now the >

$0 \parallel 1$

Last, the OR operator

1

Example 2

$6 + 7 >= 12 \&& (3+4) > 2 * 4$

The () is primary and will be performed first. 3 and 4 are added.

$6 + 7 >= 12 \&& 7 > 2 * 4$

Multiplication now has the highest precedence.

$6 + 7 >= 12 \&& 7 > 8$

The addition is performed next.

$13 >= 12 \&& 7 > 8$

Now the >=
And then the >

1 && 0

Last, but not least, the AND

0

Figure 3-2

Relational and logical operators: simple examples.

the C++ syntax, let's step back and practice writing a few "if" statements based on what we do in our everyday lives. Think of something that you check, and depending on the condition (or result) it determines an action that you perform. Here are a few examples from students.

Item to check: what day is it?

if today is Monday or today is Wednesday

Go to my C++ class

else if today is Tuesday or today is Thursday

Go to my math class

else (any other day of the week)

Do not go to school

Item to check: temperature forecast

if the temp is going to be 45 or less

Wear my heavy jacket

else if the temp is going to be 60 or less

 Wear my sweater

else

 Don't take any extra clothing

These examples show how sometimes we don't have to do anything:

Item to check: gas gauge in the car

if the gauge is less than one-quarter of a tank

 Stop and get gas

Item to check: dog's water bowl

if the dog's water bowl is dirty or if the dog's bowl is less than one-third full

 Clean and refill the bowl

The last two examples show how we do one thing or the other

Item to check: milk in the refrigerator

if the milk is still good

 Have cereal for breakfast

else

 Have toast for breakfast

Item to check: what time is it?

if the time is earlier than 7AM

 Stay in bed

else

 Get up

The general form of the *if* statement is shown below:

```
if(Condition)
{
    //These statements are executed if Condition is true or
    //skipped if Condition is false.
}
```

If the condition is true, the statements within the braces are executed. If the condition is false, the statements are skipped. For example, in the code below, gas tank level is less than 0.25, the condition is true, so the phrase "Get gasoline" is written to the screen. Examine Figure 3-3 and note how the diagram on the right has a diamond shaped box for an illustration.

```

float gasTank;
gasTank = obtain gas level from Gas Gauge
if(gasTank < 0.25)
{
    cout<< "Get Gasoline";
}

cout<< "\n Drive Safely";

```

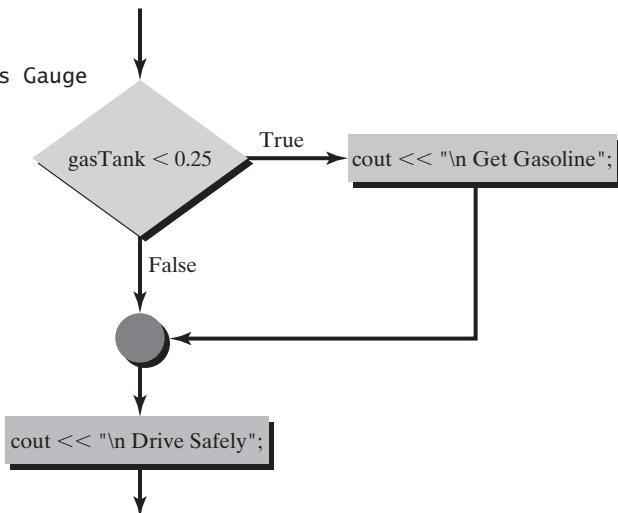


Figure 3-3
if statement.

```

float gasTank;
//we ask the GasGauge object to give us the tank level
gasTank = GasGauge.getValue();
if(gasTank < 0.25)
{
    cout << "\n Get gasoline.";
}

```

The braces do not have to be on new lines. The following format is also correct:

```

if(gasTank < 0.25){
    cout << "\n Get gasoline.";
}

```

If there is only one statement to be executed, the braces are not required. The previous sample of code can be written like this:

```
if(gasTank < 0.25) cout << "\n Get gasoline.;"
```

or like this:

```

if(gasTank < 0.25)
    cout << "\n Get gasoline.";

```

The braces are needed if more than one statement is to be executed, such as:

```

if(gasTank < 0.25)
{
    cout << "\n Get gasoline.";
    cout << "\n Wash your windows while you are at it.";
}

```

This code shows an *if* statement where the angle is checked to see if it is between 0 and 90 degrees. If the angle is within this range, we convert the angle to radians. If the angle is outside this range, the equation is skipped.

```
if(angle >= 0.0 && angle <= 90.0)
{
    radians = angle * 3.14159265/180.0;
}
```

***if-else* Statements**

For situations where something must be done if the condition is true and something else if the condition is false, C++ provides us with ***if-else*** statements. Here is the general format:

```
if(Condition)
{
    //these statements done if Condition is true
}
else
{
    //these statements done if Condition is false
}
```

In our real-world examples above, we could write our milk example like this using a boolean variable for the state of the milk. Note: programmers often follow the convention of placing a lowercase “b” in front of boolean variables to indicate (or remind the programmer) that the variable will hold either true or false values.

```
bool bMilkIsGood;           //can be either true or false
//sniff milk to determine if milk is still good

if(bMilkIsGood == true)
{
    cout << "\n Cereal for breakfast."
}
else
{
    cout << "\n Toast for breakfast."
}
```

Once again, the braces are needed if more than one statement is executed. If only one statement is executed, the braces are not needed, as shown here:

```
if(bMilkIsGood == true) cout << "\n Cereal for breakfast.";
else cout << "\n Toast for breakfast.;"
```

or

```
if(bMilkIsGood == true)
    cout << "\n Cereal for breakfast.;"
```

```
else
    cout << "\n Toast for breakfast.;"
```

In this next example we check whether a number is positive or not positive (zero or negative). We need to check only if the number is positive, because any values that are zero or negative will fall into the else statement. Figure 3-4 shows the flow of program statements.



```
if(number > 0)
{
    cout << "\n The number is positive!";
}
else
{
    cout << "\n The number is zero or negative!";
}
```

Since we are executing only one statement for each condition, this code can also be written as:

```
if(number > 0)
    cout << "\n The number is positive!";
else
    cout << "\n The number is zero or negative!";
```

```
int number;
// assume number now has a value
if( number > 0)
{
    // first set
    cout << "\n The number is positive!";
}
else
{
    // second set
    cout << "\n The number is zero or negative!";
}

cout << "\n Good Bye!";
```

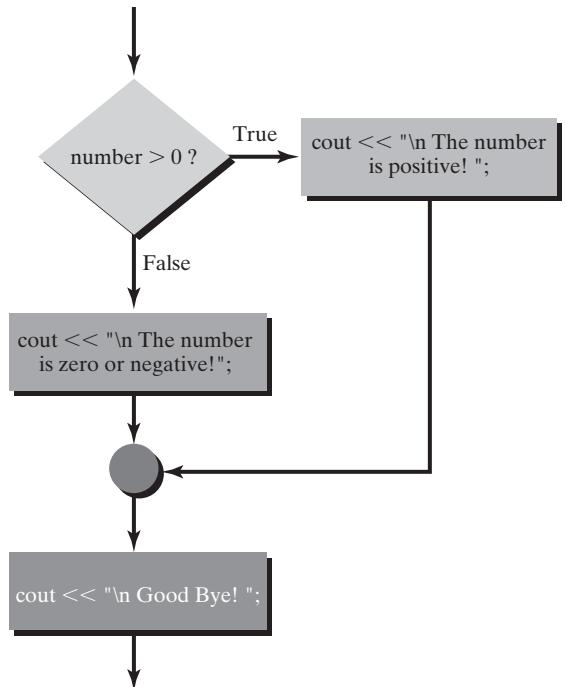


Figure 3-4
if-else statement.

It is strongly recommended that C++ programmers always use braces to enclose statements in their *if-else* statements. It is permissible not to use the braces when an *if* statement has a single statement. For any compound *if-else* or *if-else if* format, always use the braces!

Troubleshooting: Using Braces with *if* statements

A veteran C++ programmer tells the story of spending more than a day trying to find a bug in his optics design code. The error was due to having no braces in an *if-else* statement that had multiple lines in the else portion. The code in his program had been written like this:

```
//Actual manner in which the code was written.  
if(optics_condition)  
    some_optics_statement;          //Line 1  
else  
    a_different_optics_statement; //Line 2  
    a_third_optics_statement;     //Line 3
```



In this code, if the *optics_condition* is true, he wants Line 1 to be performed. If the condition is false, he wants to skip Line 1 and execute Lines 2 and 3. In fact, the following statements show how C++ interprets the code:

```
// How it is run in C++  
if(optics_condition)  
    some_optics_statement;          //Line 1  
else  
    a_different_optics_statement; //Line 2  
  
a_third_optics_statement;        //Line 3
```

This code worked well when the *optics_condition* was false, because Line 1 was skipped and Lines 2 and 3 were executed as planned. The problem occurred when the condition was true. Here, Line 1 was executed, then the *else* and Line 2 were skipped, but Line 3 was executed. (Remember, when no braces are used, C++ is built to assume that just the single statement is associated with the *if-else* statements.) By executing Line 3 accidentally when the condition was true, his program worked incorrectly. He had unhappy users, and it cost him a day to track down the bug! The code should have been written like this:

```
if(optics_condition)  
{  
    some_optics_statement;          //Line 1  
}  
else  
{  
    a_different_optics_statement; //Line 2  
    a_second_optics_statement;   //Line 3  
}
```

***if-else if-else* Statements**

The C++ programmer can cascade a series of condition-checking if statements by incorporating the *else-if* structure. The basic format of the *if-else if* statement is:

```
if(Condition1)
{
    //Condition 1 statements
}
else if(Condition2)
{
    //Condition 2 statements
}
else
{
    // else statements
}
// Rest of Program statements
```

Figure 3-5 illustrates the flow of code as these statements are executed. The program checks the first condition and, if it is true, executes the Condition1 statements that follow the *if* statement. Once these statements are completed, program control jumps to the Rest of Program statements. If Condition1 is false, then Condition2 is checked. If Condition2 is true, the Condition2 statements are performed and then the control jumps to the Rest of Program statements. Once the program has found one true condition and executes the statements for that condition, program control jumps to the statement at the end of the *if* block. If none of the conditions are true, the statements in the *else* block are performed.

Two of our real-world examples demonstrate the *if-else if* structure. We could write the daily school example like this:

```
if(today == "Monday" || today == "Wednesday")
{
    cout << "\n Go to C + + class.";
}
else if(today == "Tuesday" || today == "Thursday")
{
    cout << "\n Go to math class.";
}
else
{
    cout << "\n Do not go to school.";
}
```



The *else* block of code catches all the other days of the week, so we do not need to write a statement that checks if the day is Friday, Saturday, or Sunday.

Note the following about the *else* statement: (1) It is not necessary to have an *else* block; you may have an *if* statement, an *if-else if* statement block, or series of *if-else if-else* if statements. (2) You may have only one *else* statement in an *if* block.

```

if (Condition 1)
{
    //First set of statements
}

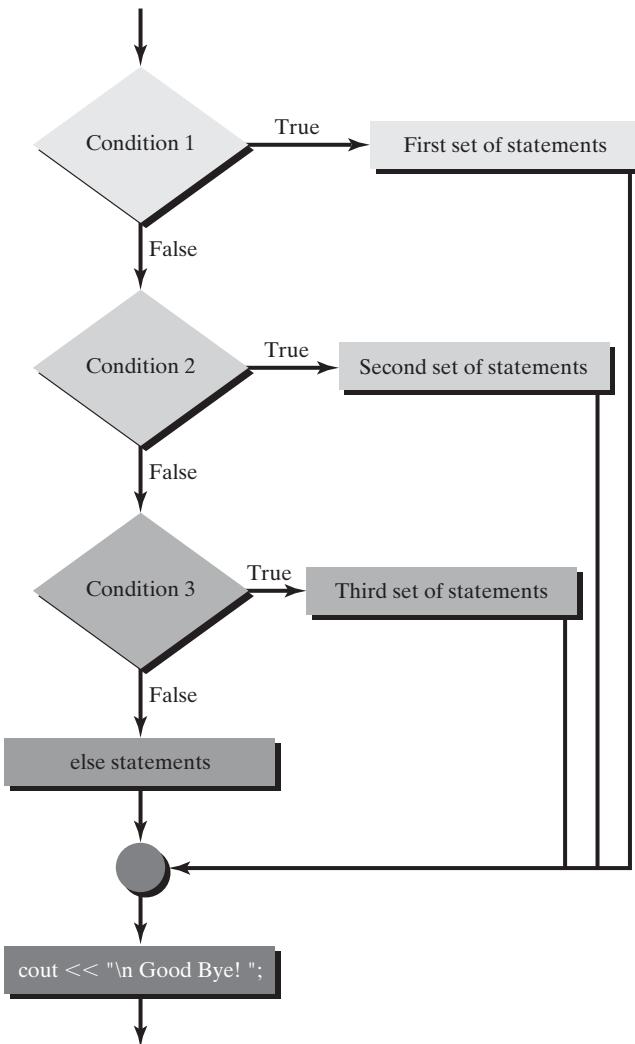
else if (Condition 2)
{
    // Second set of statements
}

else if (Condition 3)
{
    // Third set of statements
}

else
{
    // else statements
}

cout << "\n Good Bye!";

```

**Figure 3-5**Cascading *if-else if* statements.

We could re-write the temperature example so that we determine if we need a jacket or sweater. If the temperature is not going to drop below 60, we don't have to do anything, like this:

```

if(temperatureForecast < 45)
{
    cout << "\n Take your jacket!";
}

else if(temperatureForecast < 60)
{
    cout << "\n Take your sweater!";
}

```

In the following sample code, we check to see if a number is positive, zero, or negative. If the number is not positive and it is not zero, it has to be negative.

```
//Example check to see if a number is positive, zero, or negative.
//Efficient use of if else if statements
int number;
//assume number is assigned a value here
if(number > 0)
{
    cout << "\n It is positive!";
}
else if(number == 0)
{
    cout << "\n It is zero!";
}
else
{
    cout << "\n It is negative!";
}
```

Inefficient Programming Techniques

Beginning programmers sometimes forget that program control falls into the *else* statements if none of the conditions are true. They make extra work by either checking all the possible cases or checking each case individually. The following two examples have the same programmatic results but incorporate unnecessary checking:

```
//Example: check if a number is positive, zero, or negative.
//This uses an extra (unnecessary) else if statement.
if(number > 0)
{
    cout << "\n It is positive!";
}
else if(number == 0)
{
    cout << "\n It is zero!";
}
else if (number < 0)           // this check is not necessary
{
    cout << "\n It is negative!";
}
```



The second example uses three independent *if* statements. It is the most inefficient way to write software because it guarantees that the program executes all three *if* statements. If the number is positive, there is no need to perform the zero or negative checks!

```
//Example: check if a number is positive, zero, or negative.
//Uses three separate if's when a set of if-else's will do the trick!
```

```
if(number > 0)
{
    cout << "\n It is positive!";
}
if(number == 0)
{
    cout << "\n It is zero!";
}
if (number < 0)
{
    cout << "\n It is negative!";
}
```

if-else: This Old Man Program Example

We can use a popular children's rhyme and C++ together to illustrate how a program can use a set of if else statements efficiently and perform error checking as well. We see the first four lines of the famous "This Old Man" nursery rhyme. In case you have forgotten it, here it is again:

*This old man, he played one, he played knick-knack with his thumb.**

*This old man, he played two, he played knick-knack with my shoe.**

*This old man, he played three, he played knick-knack on my knee.**

*This old man, he played four, he played knick-knack at my door.**

**Chorus: With a knick-knack, paddy whack, give the dog a bone, this old man came rolling home.*

Program 3-1 asks the user to enter an integer. If it is between one and four, we write out the appropriate knick-knack information. If the user enters any other number, we write an error message. Using an *if-else if-else* block of statements allows invalid entries to be trapped in the else condition. If we code this with a series of individual *if* statements, we need to do separate checking for any values less than one and greater than four. By using the *if-else if-else* format, error trapping is easy.

Program 3-1

```
1 //Knick-Knack Example program with if else if
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int number;
9     cout << "\n Please enter an integer for knick-knacking. ";
10    cin >> number;
11    cout << "\n He played knick-knack ";
```

```

12      if(number == 1) //write out knick-knack information
13      {
14          cout << "with his thumb. \n";
15      }
16      else if(number == 2)
17      {
18          cout << "with my shoe. \n";
19      }
20      else if(number == 3)
21      {
22          cout << "on his knee. \n";
23      }
24      else if(number == 4)
25      {
26          cout << "at the door. \n";
27      }
28      else           // error check, any other number is not valid
29      {
30          cout << "\n Whoa! He doesn't play knick-knack there!\n\n";
31      }
32      return 0;
33  }

```

Output with input value of 3

Please enter an integer for knick-knacking. 3
He played knick-knack on his knee.

Output with input value of 7

Please enter an integer for knick-knacking. 7
He played knick-knack
Whoa! He doesn't play knick-knack there!

Nested *if-else* Statements

nested *if* statements

one set of *if* statements located inside another set of *if* statements

It is possible to ***nest*** *if* statements within *if* statements. A nested *if* statement is simply an *if* statement as part of the code inside portions of another *if* statement, like this:

```

if(Condition)           //first condition
{
    if( Another Condition ) //this is the nested if statement
    {
        // statements
    }
    // more statements
}

```

In C++, it is possible to nest most types of statements. For our real-world example, we can re-write the dog bowl example so that we check if the dog's bowl is

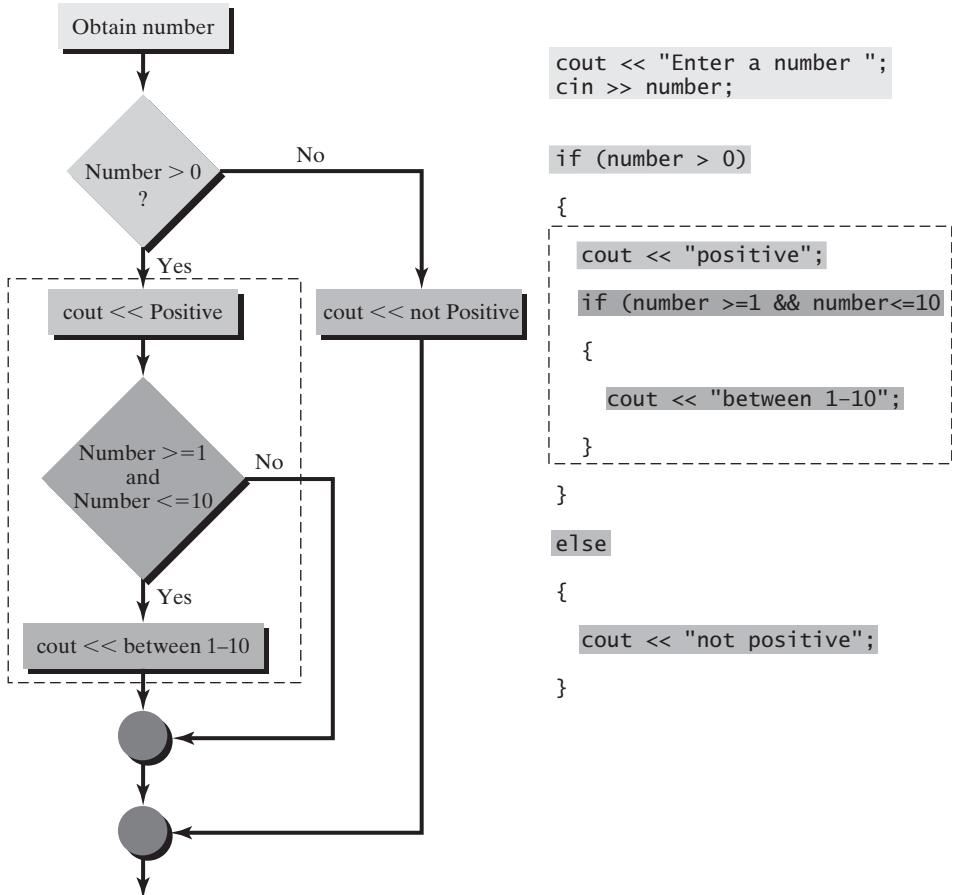
less than one-third full, and then check if the bowl is dirty. The nested statements look like this:

```
string dogBowlState;
float dogBowlLevel;
//place values in these two variables
if(dogBowlLevel < 0.333)           //first thing to check
{
    if(dogBowlState == "dirty") //next thing to check
    {
        cout << "\n Need to wash the bowl.";
    }
    cout << "\n Fill the bowl with water.";
}
```

Here is another nested *if* block. Rolando came out one morning and his car wouldn't start. He knew how to check his car's battery and gas gauge. Finding no problem, he then called his roadside assistance organization for a tow to the dealership. Notice how this sequence of events can be coded in a nested *if* block.

```
//Rolando is trying to drive to work.
if(car won't start)
{
    if(gas gauge is on empty)
    {
        cout << "\n Need to get gasoline!";
    }
    else if( battery is dead)
    {
        cout << "\n Need to get a jump!";
    }
    else
    {
        cout << "\n Call for a tow.";
    }
}
else
{
    cout << "\n Drive to work";
}
```

Beginning programming students sometimes have trouble taking a problem from a flow diagram into C++ code. In Program 3-2, we ask the user to enter a whole number. If the number is zero or negative, we just report that. If it is positive, we report only if the number is between 1 and 10. We use a nested *if* structure to accomplish this task. Figure 3-6 illustrates how the flow diagram maps into C++ code.

**Figure 3-6**

The flow diagram on the left is color-coded to match the C++ statements on the right.

Program 3-2

```

1 //Nested if statements sample program
2
3 #include <iostream>
4 using namespace std;
5
6
7 int main()
8 {
9     int number;
10
11     cout << "\n This program will tell you if your number is "
12             << "\n positive and between 1 and 10. ";
13
14     cout << "\n Please enter a whole number:   ";
15     cin >> number;
    
```

```

16     if(number > 0) //positive number
17     {
18         cout << "\n The number is positive. " << endl;
19
20         if(number > = 1 && number < = 10)
21         {
22             cout << " It is between 1 and 10." << endl;
23         }
24     }
25
26 }
27 else
28 {
29     cout << "\n The number is not positive." << endl;
30 }
31 return 0;
32 }
```

Output with input value of 5

This program will tell you if your number is positive and between 1 and 10.

Please enter a whole number: 5

The number is positive.

It is between 1 and 10.

Output with input value of 48

This program will tell you if your number is positive and between 1 and 10.

Please enter a whole number: 48

The number is positive.

Output with input value of -68

This program will tell you if your number is positive and between 1 and 10.

Please enter a whole number: -68

The number is not positive.

The ? Operator

The C++ language has a ? operator called a ternary operator. A **ternary operator** requires three operands. This ? operator functions like the *if-else* statement, with some restrictions. Let's review an *if-else* example:

```

float temperature;
string AirConditioner
//assign a value into temperature
if(temperature > 80)
    AirConditioner = "on";
else
    AirConditioner = "off";
```

ternary operator

an operator that requires three operands

The temperature is checked, and if it is higher than 80 degrees, we wish to turn on the air conditioner.

The format of the ? operator is shown below:

```
Expression1 ? Expression2 : Expression3;
```

The Expression1 is evaluated and if it is true, Expression2 is assigned in Expression1. If Expression1 is false, Expression3 is assigned. The following statements are equivalent to the if else statements above:

```
float temperature;
string AirConditioner;
//assign value into temperature
AirConditioner = temperature > 80 ? "on" : "off";
```

A second example with the ? operator illustrates a quick way to set a value to either true or false. First, let's show the code in an *if-else* format. We set the value Hurricane to true if the wind speed is greater than 75 miles per hour.

```
int windSpeed;
bool bHurricane;
// code sets value for windSpeed
if(windSpeed > 75)
    bHurricane = true;
else
    bHurricane = false;
```

The ternary operator provides a way to perform this check and assignment in one line.



```
int windSpeed;
bool bHurricane;
// code sets value for windSpeed
bHurricane = windSpeed > 75 ? true : false;
```

The ? operator does not produce easy-to-read code. It is presented here for completeness, but it is not a format that we encourage beginning C++ programmers to use.

3.3 ***switch Statement***

The *switch* statement provides an alternate method for doing a series of condition checks and statement executions. It is built for checking the value of an integer expression. This means that a switch can be used to check either integers or characters. (Remember that characters are stored as integer values, see Appendix D.) The basic form of the switch is shown below:

```
switch(variable)
{
    case value1:
        //statements 1
```

```
break;
case value2:
    //statements 2
    break;
//have all case statements before default
default:
    //statement n
} //close brace
```

In a *switch* statement, the value of the variable is examined (an expression may be used and evaluated too). If the value is one of the values in the case statements, the associated statements are performed. The *break;* statement causes the program to jump to the closing brace. If none of the case values are found, the statements in the default statement are performed. The Knick-Knack program can be written by using a *switch* statement instead of the *if* statements. See Program 3-3.

Program 3-3

```
1 // Knick-knack Example program with switch
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int number;
9     cout << "\n Please enter an integer for knick-knacking ";
10    cin >> number;
11
12    cout << "\n He played knick-knack ";
13    switch(number) //write out area that will be knick-knacked.
14    {
15        case 1:
16            cout << "with his thumb. \n";
17            break;
18        case 2:
19            cout << "with my shoe. \n";
20            break;
21        case 3:
22            cout << "on his knee. \n";
23            break;
24        case 4:
25            cout << "at the door. \n";
26            break;
27        default:
28            cout << "\n whoa! He doesn't play knick-knack there! \n";
29    }
30
31    return 0;
32 }
```



Output with input value of 1

```
Please enter an integer for knick-knacking 1
He played knick-knack with his thumb.
```

The case statements in the *switch* block can be used to check a variety of values. The values do not have to be a sequence of integers (as shown in the Knick-Knack program). Program 3-4 illustrates how the case statements can be used for different integer values.

Program 3-4

```
1 //Famous Year Program with switch
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int year;
9     cout << "\n Please enter your favorite year    ";
10    cin >> year;
11
12    cout << "\n Your year: " << year << " is famous because ";
13    switch(year) //check for a famous year
14    {
15        case 1920:
16            cout << "women were allowed to vote!\n";
17            break;
18        case 1776:
19            cout << "the Declaration of Independence was drafted!\n";
20            break;
21        case 1969:
22            cout << "Neil Armstrong walked on the moon!\n";
23            break;
24        default:
25            cout << "\n ...oh, not sure what happened in your year.\n";
26    }
27
28    return 0;
29 }
```

Output with input value of 1776

```
Please enter your favorite year 1776
```

```
Your year: 1776 is famous because the Declaration of Independence was
drafted!
```

The *switch* can be used for checking character data too. Program 3-5 shows a *switch* statement checking the value of the letter that the user enters. Note too

how you can use multiple case statements, such as checking to see if the letter is an ‘a’ or an ‘A.’

Program 3-5

```
1 //A program that asks the user to enter the
2 //letter of his favorite fruit. We use a switch
3 //statement to determine the value of the letter.
4
5 #include <iostream>           //for cout, cin
6
7 using namespace std;
8
9 int main()
10 {
11
12     string fruit;
13
14     cout << "\n I'm going to try and guess your favorite"
15         << " fruit. \n\n Please enter the first letter"
16         << " of your favorite fruit: ";
17
18     char firstLetter;
19     cin >> firstLetter;
20
21     switch (firstLetter)
22     {
23         case 'a': case 'A':
24             cout << "\n Is it apple? " << endl;
25             break;
26         case 'b': case 'B':
27             cout << "\n Is it banana? " << endl;
28             break;
29         case 'c': case 'C':
30             cout << "\n Is it cantaloupe? " << endl;
31             break;
32         default:
33             cout << "\n I can't guess your favorite fruit. ;-)"
34                 << endl;
35     }
36
37     return 0;
38 }
```

Output with input value of 'b'

I'm going to try and guess your favorite fruit.
Please enter the first letter of your favorite fruit: b
Is it banana?



Troubleshooting: Don't Forget to Break Your Switch

Forgetting to include the break statement is a common mistake! If the programmer forgets the break statement, the program continues executing the case statements without breaking (jumping) out of the switch. For example, if you coded the Knick-Knack switch, like this, and the user entered a 2, the following would be the output:

```
cout << "\n He played knick-knack on ";
switch(number) //Incorrectly coded switch, no break statements!
{
    case 1:
        cout << "with his thumb";
    case 2:
        cout << "with my shoe";
    case 3:
        cout << "on my knee";
    case 4:
        cout << "at the door";
    default:
        cout << "\nWhoa! He doesn't play knick-knack there!";
}
```



Output with input value of 2

He played knick-knack with my shoe. on my knee. at the door.
Whoa! He doesn't play knick-knack there!

3.4 Loops in General

loop

series of C++ statements that enable the program to repeat line(s) of code until a certain condition is met

A *loop* is a fundamental tool for all programming languages. It enables the program to loop over or repeat statements. This iterative process may be set up so that the loop is executed a predetermined number of times or until a certain condition is met. Once again, let's think about loops by looking at a few real-world situations where we check a condition, and repeat a behavior or an action based on that condition. For these examples, we use the term “while” to state our condition.

Item to check: C++ program state

while the program is not working correctly

keep debugging the program

Item to check: golf ball

while the ball is on the green and the ball is not in the hole

use putter to putt the ball into the hole

Item to check: contents of cereal bowl

while there is cereal in the bowl

keep eating

The programmer, who is also a chef, gave us this example for making roux, the thickener used in gumbo:

Item to check: roux state

while roux is cooking and it is not browned

keep stirring

Sometimes our loops need to execute code an exact number of times. In this case, we can use a counter to keep track of the number of times we've performed the statements. Thinking in terms of real-world situations, our basketball player came up with this scenario:

Item to check: free throw percentage in basketball games

if her free throw percentage is less than 70%

shoot 100 free throws everyday after practice (count free throws)

Our bowler proposed this loop situation:

Item to check: bowling frame

while frame number is 1 or greater and frame number is 10 or less

keep bowling

The C++ language provides three methods for performing loops: the *for* loop, the *while* loop, and the *do while* loop. All loops in C++ have either a **loop index**, counter variable, or stopping variable, and the following steps must be taken:

1. An initial assignment for the loop counter or stopping variable.
2. A condition such that, when it is true, it will cause the loop to be executed, and when it is false, it will cause the program to stop the loop.
3. **Loop altering statement** that will adjust the counter or stopping variable.

loop index

variable used as a counter in a loop

loop altering statement

line of C++ code that changes the value of a variable used in the condition-checking decision for a loop

To Brace or Not to Brace?

All three loop formats in C++ require opening and closing braces if the loop is to execute more than one statement. No braces are required if the *for* and *while* loops execute only one statement. (The brace requirement is exactly the same as that for the *if* statements.) In this text, we use braces for our loops and encourage readers to do the same. The use of braces with loops and ifs, and indenting code within the braces, makes the code easier to read. It also aids the programmer in debugging problems.

You Can't Get Out of an Infinite Loop

An infinite loop occurs when a loop starts but the counter limit or stopping condition is never met and the loop never stops executing. (You get checked in but you can't check out of an infinite loop!) A program starts running and then seems to hang or pause forever, when actually the loop is executing. This situation often requires the programmer to use the Control+c sequence (Ctrl-c) to stop the program execution.

Recommendations! Always be sure the loop conditions are reasonable and that, once the loop starts, it will be able to stop. Also, save your program before you run it. If you have an infinite loop and you must kill the program, your recent additions to the file may not be saved automatically. (The Microsoft Visual C++ 2005 Express and .NET compilers default to saving files before running them.)

3.5 for Loop

The **for loop** is a convenient C++ statement for use when the programmer knows exactly how many times the statements must be repeated. This loop structure has the following format:

```
for(initial assignment; condition to test; loop test update)
{
    //statements are executed if condition is true
}
```

Here is an example of writing *hello world* to the screen fifty times:

```
//write hello world 50 times
int i;
for(i = 0; i < 50; ++i)
{
    cout << "\n hello world";
}
```

The **for** loop is illustrated as a flow diagram in Figure 3-7. Figure 3-8 shows another view of how the **for** loop is executed. Study both figures and see that C++ first executes the assignment statement. Here the integer “*i*,” acting as the loop counter or loop index, is assigned a value of zero. Next, the condition is checked: Is *i* less than 50? Yes, so the statement(s) within the braces are performed. At the end of the statements, the program performs the increment (it adds 1 to *i*) and checks the condition again. If the condition is true, the statements within the braces are executed again. If the condition is false, program control then goes to the statement after the closing brace.

It is possible to use the decrement operator instead of the increment operator to vary the loop variable. This loop writes the numbers 100 to 1 to the screen.

```
//write numbers 100 to 1
int ctr;
for(ctr = 100; ctr > 0; --ctr)
{
    cout << "\n The counter  =" << ctr;
}
```

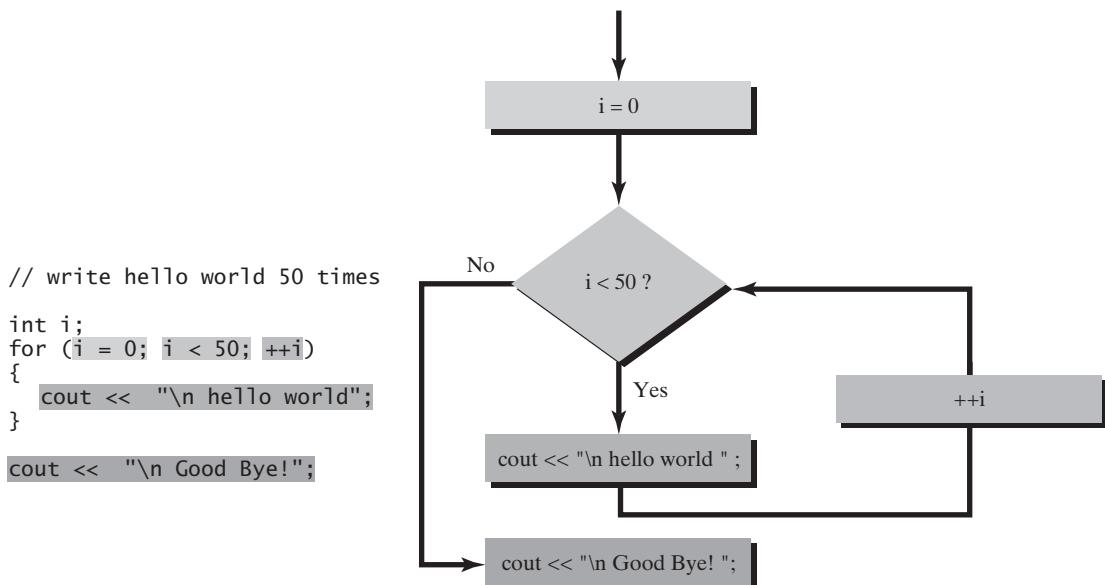


Figure 3-7
for loop.

In *for* loops, the assignment statement is performed and the condition is checked. If the condition is false, the loop does not execute, as seen here:

```

int t;
for(t = 0; t < 0; ++t) //this loop will not execute
{
    //loop statements
}
    
```

The above examples are by far the most commonly used forms of the *for* loop, but the programmer can use various forms as well as more complicated logic.

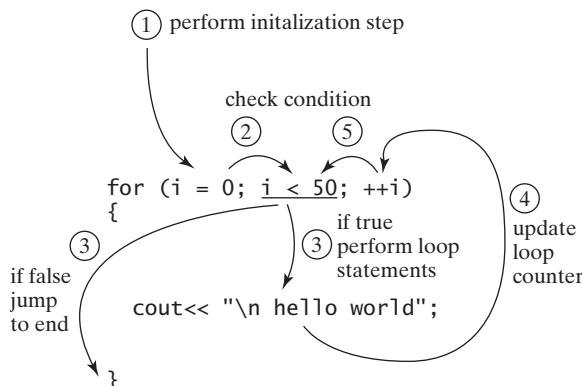


Figure 3-8
Sequence of events in a *for* loop execution.

The control for the *for* loop can be based on the value of two or more variables. The example below illustrates this situation:

```
int x,y;
for(x = 0, y = 100; x < 50 && y > 30; ++x, --y)
{
    //statements are executed if condition is true
}
```

The comma can be used to string together the initialization and increment/decrement statements. The condition may use logical operators for complicated conditional checks, but beginning programmers should avoid using this type of program code.

Do Not Alter the Loop Index

Programmers should not tinker with the loop counter inside the *for* loop. This loop structure is built to initialize and check the condition and increment if the loop statements have been performed. It is poor practice to use creative logic in a *for* loop. In the example below, the index “k” is altered inside the loop. This alteration results in an infinite loop because “k” never reaches the value of 10.



```
int k;
for(k = 1; k < 10; ++k)
{
    //loop statements
    k--;                                //<== DO NOT DO THIS!!
}
```

for Loop Examples

HowManyHellos The HowManyHellos program in Program 3-6 asks the user to enter the number of “hellos” she wishes to see. The program has a variable for the user’s value as well as a counter for the *for* loop that keeps track of the times the loop has executed.

Program 3-6

```
1  //A program to write hellos using a for loop
2
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8      int counter, howMany;
9
10     cout << "\n How many hellos would you like to see? ";
11     cin >> howMany;
12 }
```

```

13     //for loop executes howmany times
14     for(counter = 0; counter < howMany; ++counter)
15     {
16         cout << "\n Hello!";
17     }
18
19     cout << "\n That's a lot of hellos! \n";
20
21     return 0;
22 }
```

Output with input value of 5

How many hellos would you like to see? 5

```

Hello!
Hello!
Hello!
Hello!
Hello!
That's a lot of hellos!
```

Writing the Alphabet We wish to write the letters of the alphabet (in capital letters) in four rows so that the output looks as shown here:

A	B	C	D	E	F	G
H	I	J	K	L	M	N
O	P	Q	R	S	T	U
V	W	X	Y	Z		

We can use a *for* loop and the ASCII character codes to access the letters directly. For example, the letter “A” is stored as 65. To write an “A,” we cast a 65 into a character in the *cout* statement. (Consult Appendix D, “ASCII Character Codes,” to see the full range of codes.) To write the alphabet on four lines, we use a counter to keep track of how many letters we have written to the screen. Seven letters are written on three lines, with the remaining five characters on the last line. Once we have written seven letters, we write a newline character and reset the counter.

Program 3-7

```

1 //Writing the ABC's using a for loop
2
3 #include <iostream>           //for cout
4 #include <iomanip>           //for setw
5 using namespace std;
6
7 int main()
8 {
9     int letter_ctr, i;
10    cout << "\n We're going to write our ABCs \n\n";
```

```

12      letter_ctr = 0;
13      for(i = 65; i < 91; +i)           //A = 65, Z = 90
14      {
15          //we cast the integer into a character
16          cout << setw(6) << static_cast<char>(i);
17
18          letter_ctr++;           // incr letter counter
19          if(letter_ctr == 7)    // newline if we've written 7
20          {
21              cout << endl;
22              letter_ctr = 0;
23          }
24      }
25
26      cout << "\n\n There are our ABCs." << endl;
27
28      return 0;
29  }

```

Output

We're going to write our ABCs

A B C D E F G

H I J K L M N

O P Q R S T U

V W X Y Z

There are our ABCs.

3.6

while Loop

The **while loop** is needed when the programmer does not know how many times a loop is to be executed. Since the condition is checked first, it is possible that the loop will not execute. The **while** loop can also be used to perform a loop an exact number of times. The **while** loop format is shown below and is illustrated in Figure 3-9.

```

while(condition)
{
    //these statements done if condition is true
}

```

The **while** loop checks the condition and if it is true, the loop statements are performed. If the condition is false, the statements are skipped.

To avoid becoming stuck in an infinite loop, a more complete form for this loop includes an initialization statement and a loop altering statement, as shown here when we write *hello world* to the screen fifty times.

```
// write hello world 50 times
count = 0; // initialize count to zero
while (count < 50 )
{
    cout << "\n hello world";
    ++ count; // loop altering statement
}
cout << "\n Good Bye! ";
```

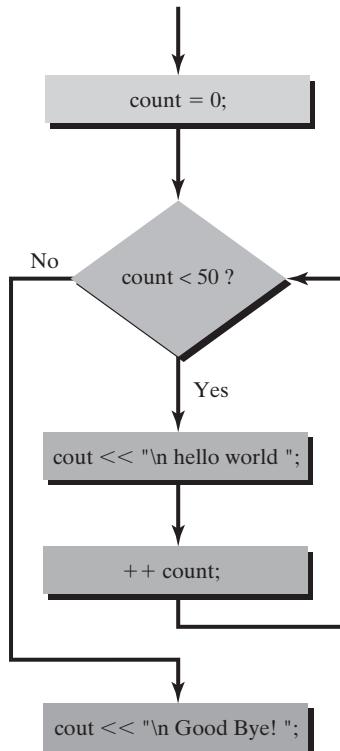


Figure 3-9
while loop.

```
int count = 0; //important to set up initial condition
while(count < 50) //condition check
{
    cout << "\n hello world"; //alter the loop counter
}
```

Re-visiting our real-world examples, we can write the bowling examples like this:

```
int FrameNumber = 1; //begin at Frame 1
while(FrameNumber >= 1 && FrameNumber <= 10)
{
    cout << "\n Keeping bowling.";
    FrameNumber++;
}
```

Here is a *while* loop inspired by the golfers in our class. Remember, *while* loops can be designed so they execute until a certain condition or conditions are met. In golf, once your ball is on the green, you keep putting until it falls into the cup (hole). If you knock your ball off the green, you need to use a different club to get it back on the green. We'll set two boolean flags to indicate the state of the ball

with regard to the green and the hole. Notice how we rely on the AND operator to check two conditions:

```
bool bBallOnGreen = true;
bool bBallInHole = false;
while(bBallOnGreen == true && bBallInHole == false)
{
    cout << "\n Keeping putting!";
    Golfer.puttTheBall();
    if(ball falls into the hole) bBallInHole = true;
    else if (ball flies off the green) bBallOnGreen = false;
}
```

while Loop Examples

In the following programmatic examples, we must remember to set up an initial condition for the loop counter. If we forget to assign the initial condition, there is no telling what might be in the memory allocated for count. Also note that there must be a loop altering statement inside the *while* loop, or we find our program stuck in an infinite loop.

A Lovely Poem Consider a program that illustrates how the *while* loop runs until the correct stopping condition is found. Remember, as long as the condition is true, the loop executes. In Program 3-8, we write out a poem and ask the user if he would like to see it again. As long as the user wants to see it, we keep writing it.

Note that we initialize the answer to “yes” so that the loop runs the first time. After we write out our lovely poem, we ask the user if he would like to see it again. Once the user has entered his answer, the program then checks the condition in the *while* statement. As long as the condition is true, the loop executes.

Program 3-8

```
1 //while loop and Poetry program
2
3 #include <iostream>           //for cout, getline
4 #include <string>            //for string object
5 using namespace std;
6
7 int main()
8 {
9
10     string answer = "yes";   //initialize answer to yes
11
12     cout << "\n Here is my lovely poem. ";
13
14     while(answer == "yes")   //keep going until not yes
15     {
16         cout << "\n Roses are red, violets are blue" <<
17             "\n I love C ++, how about you?";
18
19         cout << "\n Would you like to see my poem again? yes/no ";
```

```
20         getline(cin, answer);
21     }
22
23     cout << "\n Did you enjoy my poem?" << endl;
24     return 0;
25 }
```

Output with answering yes once, then no

Here is my lovely poem.

Roses are red, violets are blue

I love C++, how about you?

Would you like to see my poem again? yes/no yes

Roses are red, violets are blue

I love C++, how about you?

Would you like to see my poem again? yes/no no

Did you enjoy my poem?

HowManyHellos Program 3-9 illustrates how an expression with variables can be used in the conditional portion of a *while* loop. The HowManyHellos program is rewritten by means of a *while* loop. Take a moment to review the *for* loop version of this program. Note that the counter must be initialized and incremented in separate lines of code—as opposed to the *for* loop, where these statements are all performed on one line.

Program 3-9

```
1 //Writing hellos with a while loop
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int counter, howMany;
9
10    cout << "\n How many hellos would you like to see?      ";
11    cin >> howMany;
12
13    counter = 0;
14
15    while(counter < howMany)    //loop executes howMany times
16    {
17        cout << "\n Hello!";
18        ++counter;
19    }
20
21    cout << "\n That's a lot of hellos!  \n";
22
23    return 0;
24 }
```

Output with input value of 5

```
How many hellos would you like to see? 5
```

```
Hello!
Hello!
Hello!
Hello!
Hello!
```

That's a lot of hellos!

3.7 do while Loop

The third type of loop structure in C++ is the ***do while loop***. It is very similar to the ***while*** loop except that the condition check is performed at the end of the loop. The loop statements are always performed at least once—unlike the ***for*** and ***while*** loops, where the condition must be true before the loop is executed. Refer to Figure 3-10.

```
do
{
    //loop statements
} while(condition);
```

```
// write hello world 50 times
count = 0; // initialize count to zero
do
{
    cout << "\n hello world";
    ++ count; // loop altering statement
} while (count < 50);

cout << "\n Good Bye! ";
```

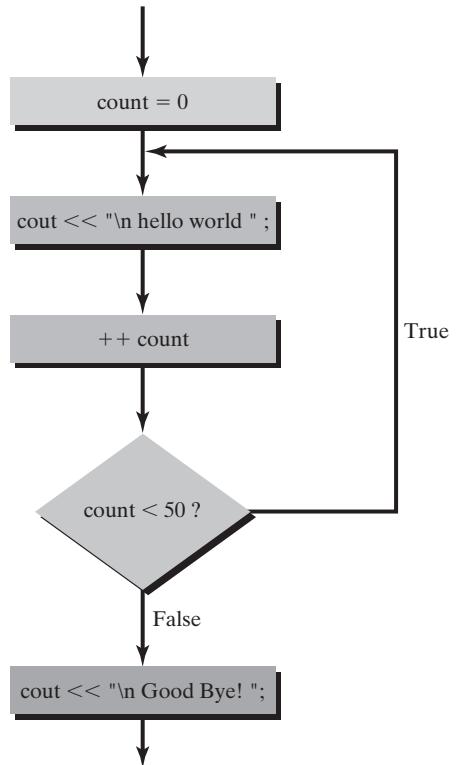


Figure 3-10
do while loop

The program executes the loop statements and then checks the condition. If the condition is true, the program control returns to the *do* statement and executes the loop statements again. As in the *while* and *for* loops, you must initialize your counter or stopping condition and have loop altering statements. We can write *Hello World* fifty times using a *do while* loop.

```
int count = 0;
do
{
    cout << "\n Hello world";
    count++;
} while(count < 50);
```

***do while* Examples**

A Lovely Poem Throughout this book, the problem descriptions usually request that the user be allowed to loop back to the beginning of the program if she wishes. This request allows the user to continue to loop through the main portion of the program as many times as desired. The *do while* loop is perfect for this type of application.

Our poetry program appears again in Program 3-10. This time the program makes use of the *do while* loop format. The statements within the loop are executed once and then the conditional statement is evaluated.

Program 3-10

```
1 //do while loop and Poetry
2
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 int main()
8 {
9     string answer; /*no need to initialize since we'll ask the user
10                before we check the condition */
11
12     cout << "\n Here is my lovely poem. ";
13
14     do
15     {
16         cout << "\n Roses are red, violets are blue" <<
17             "\n I love C++, how about you?";
18
19         cout << "\n Would you like to see my poem again? yes/no ";
20         getline(cin,answer);
21
22     } while(answer == "yes"); //keep going until not yes
23 }
```



```

24     cout << "\n Did you enjoy my poem?" << endl;
25
26     return 0;
27 }
```

Output with answering yes once, then no

Here is my lovely poem.

Roses are red, violets are blue

I love C++, how about you?

Would you like to see my poem again? yes/no yes

Roses are red, violets are blue

I love C++, how about you?

Would you like to see my poem again? yes/no no

Did you enjoy my poem?

3.8

Jump Statements

jump statement

one of four statements that causes the program control to jump to another location

C++ has four ***jump statements*** that cause the program control to “jump” to another location in the code without checking a condition. The four statements are *break*, *continue*, *return*, and *goto*. The *break* and *continue* statements should be used in loop statements; we investigate them further here. (We’ve already seen the *break* in the *switch* statement.) The *return* is used to exit from inside a function; we’ll see it used in the next chapter. The *goto* statement allows program control to jump to a labeled statement in your code. Using *goto* statements can make the code unreadable, and hence, not a statement that you should incorporate in your code. (We’ll leave it to the reader to investigate the *goto* on his own.)

break**break**

a statement that causes control to jump to end of block of code

The ***break statement*** can be used in two situations. In a *switch* statement, *break* is used to terminate a case condition. In the Favorite Fruit program (page 121) we see the *break* statement within each case. The *break* causes the program control to jump to the end of the *switch* statement.

```

switch (firstLetter)
{
    case 'a': case 'A':
        cout << "\n Is it apple? " << endl;
        break;           //== this break makes control jump to ....
    case 'b': case 'B':
        cout << "\n Is it banana? " << endl;
        break;
    case 'c': case 'C':
        cout << "\n Is it cantalope ? " << endl;
        break;
```

```
default:
    cout << "\n I can't guess your favorite fruit.  ;)" << endl;
}           //<== here, exiting the switch statement
```

The *break* statement can also be used to immediately stop a loop. That is, if the program reaches a *break* statement inside a loop, program control jumps to the line of code following the loop. The loop condition is not checked. In this sample code, we give the user five tries to enter the magic password (*letMeIn*). If the user enters the correct password, we break out of the loop.

```
string password;
for(int i = 0; i < 5; ++i)           //give the user 5 tries
{
    cout << "\n Enter the magic password" << endl;
    getline(cin,password);
    if(password == "letMeIn")
        break;                      //<== now jump out of loop
}
//<== end of loop, control goes to line following this brace.
```

continue

The ***continue statement*** is used in loops statements and it forces the next iteration of the loop to occur. When the program control reaches a *continue* statement in a *for* loop, control jumps to the increment statement, then checks the conditional statement in the loop. If a *continue* is reached in a *do while* or *while* loop, program control jumps to the conditional statement. In the code samples below, we ask the user to enter a whole number. If the user's number is positive, we add it to the sum value. If it is zero or negative, we jump back to the top of the loop, and proceed to ask the user for another number.

```
int i = 0, number, sum = 0;
cout << "\n Hi, we need 25 positive values for your sum.\n";
while( i < 25)
{
    cout << "\n Enter the a whole number" << endl;
    cin >> number;
    if(number <= 0)
        continue;                  //go back to start of loop
    else
    {
        cout << "\n That is your" << i+1 << "positive number.";
        ++i;
        sum = sum + number;
        cout << "\n Your sum so far is" << sum;
    }
}
```



3.9 Troubleshooting

Five Common Mistakes

Relational and logical operators can be tricky! C++ has strict syntactical rules for writing these statements. Sometimes a beginning programmer gets stuck because he makes one (or more) of these common mistakes. Study these five errors.

First Mistake: Counter > 5 && < 10

same as operator ==
an operator that compares the two operands to determine if they are the same

assign operator =
an operator that copies the value of the right operand into the left operand

Suppose a programmer needs to check whether a counter value is greater than 5 and less than 10. Each phrase of a conditional statement must be written completely, meaning that you should write, “Counter is greater than 5 and Counter is less than 10,” not “Counter is greater than 5 and less than 10.” Remember, relational operators have a higher precedence than logical operators—extra parentheses are not required. The correct way to write this statement is:

```
if(Counter > 5 && Counter < 10) //correct!
```

The following statement shows a common mistake when writing complicated statements:

```
if(Counter > 5 && < 10) //incorrect way!!
```

The compiler does not expect to find the “<” symbol right after the “**&&**” symbol and gives an error on this statement:

```
test.cpp(13) : error C2059: syntax error : '<'
```

Here is another tricky error. A programming student was attempting to check the condition of a variable to see if it was either 0 or 1. He wrote his statement as follows:

```
if(x == 0 || 1) // incorrectly written
```

When he ran his program, the statement was never found to be false. Why? The OR requires either the left side or the right side to be 1 (true) to return true. Since the right side of the OR operator was a 1, the statement was always evaluated to be true. The correct way to write this statement is:

```
if(x == 0 || x == 1) //correct
```

Second Mistake: A = B Is Not A == B

The relational operator, **==**, evaluates the two operands to see if they have the same value. The assignment operator, **=**, assigns the value on the right to the variable on the left. Programmers either forget this or mistype the statement. For example:

```
while(a = b) // assign b into a
```

```
while(a == b) // checking to see if a and b have same value
```

This simple error can cause a huge amount of programming grief! Check out the code below. Can you figure out why this loop won't stop?

```
int goAgain;
do
{
    // Programming details have been left out.
    cout << "\n Would you like to go again? 1 = yes 0 = no";
    cin >> goAgain;
}while(goAgain = 1);
```

Recommendation! Programmers should get in the habit of calling the == operator the “same as” operator and the = operator the “assign” operator. Too often students use the word equals for both concepts and inadvertently use the = operator when they mean the == operator.

Third Mistake: *floats* Are Not the Same As *ints*

When you are evaluating floating point or double variables to determine if the values are the same, using the same as operator does not guarantee accurate results. Integer values are stored precisely in memory and will evaluate correctly when using the == operator. Float and double variables have a small inaccuracy due to the manner in which decimal precision values are stored. Program 3-11 illustrates this problem. It shows the wrong and right way to perform this comparison. We use the absolute value function, *fabs*, located in the *<cmath>* library. The *fabs* absolute value function works on *floats* and *doubles*, and returns the positive value of a number. (That is, if we take the absolute value of -6.354, we get 6.354.) Hint: run program 3-11 in the debugger and watch the value for sum.

Program 3-11

```
1 //Program that shows the right way and wrong way to
2 //compare floats and doubles for exact value.
3
4 #include <iostream>                      //for cout
5 #include <cmath>                          //for fabs function
6 using namespace std;
7
8 int main()
9 {
10
11     float sum = static_cast<float>(0.0);
12     float nickel = static_cast<float>(0.05);
13     float dollar = static_cast<float>(1.00);
14
15     //add 20 nickels, result should be 1.00
16     for(int i = 0; i < 20;  ++i)
```

```

17      {
18          sum = sum + nickel;
19      }
20
21
22      //Here is the wrong way to check a float value.
23      if(sum == dollar)
24      {
25          cout << "\n First check: Sum is $1.00 ";
26      }
27      else
28      {
29          cout << "\n First check: Sum is not $1.00 ";
30      }
31
32
33      //Here is the right way to check a float value.
34      //Check to see that the absolute value of the
35      //difference is a small value. This works!
36
37      if( fabs(sum - dollar) < 0.00001)
38      {
39          cout << "\n Second check: Sum is $1.00 " << endl;
40      }
41      else
42      {
43          cout << "\n Second check: Sum is not $1.00 " << endl;
44      }
45
46      return 0;
47  }

```



STOP

Output

First check: Sum is not \$1.00
 Second check: Sum is \$1.00.

Fourth Mistake: Semicolons and Braces

One of the nastiest little bugs that strike a program is caused by the programmer accidentally placing a semicolon after the parentheses, like the examples shown here. In both cases, the compiler believes there is only one statement to be executed and ignores the braces. What is wrong with this code? What will happen when it runs?

```

int count = 100;
if(count < 50);                      //<== Oh No! ;
{
    cout << "\n Count is less than 50.";
}

```

Or how about this code? What will happen when it runs?

```
int count = 0;
while(count < 6 );           //<< uh oh, ;
{
    cout << "\n Hi there.";
    ++count;
}
```

For the first example, the statements within the *if* statement braces are executed because the “empty” statement is executed only if the condition is true. (We’d see “Count is less than 50” in the output window.) In the case of the *while* loop, the program gets stuck in an infinite loop because the empty statement is executed until the condition is false—which never happens. In the case of the *if* statement, the Microsoft Visual C++ compilers warns the programmer that an empty statement has been found:

```
warning C4390: ';' : empty controlled statement found; is this the intent?
```

Remember that it is legal in C++ to have a single statement executed after a conditional expression. Always take a few minutes to read each warning. And always look at your condition statements to be sure the loop or *if* statement is in the desired format.

Fifth Mistake: Misplaced *else*, Illegal *else*, Unexpected End of File

*Style is important when programming *if* statements, *switch* statements, and loops!* Beginning programmers often view the requirement of aligning the opening and closing braces and indenting code as an after-the-fact task. If the program has complicated logic, however, there will be many pairs of braces in the source code. You will not be able to see all the braces on the screen while editing. It is good practice to develop the habit of aligning the braces and indenting the code within the braces as you enter your C++ statements.

The most common mistake programmers make is not having complete sets of braces when using the *if-else*, *switch*, and *loop* statements. The compiler does its best to match up the opening and closing braces; it will attempt to report the location where it suspects the missing brace should be placed.

To gain a feel for debugging code with poor style, Program 3-12 is a shortened version of the Knick-Knack program. This code will generate the two compiler errors shown below it. Can you spot the problem?

Program 3-12

```
1 //Short Knick-Knack Program with compiler errors
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
```



```

7  {
8  int number;
9  cout << "\n Please enter an integer for knick-knacking";
10 cin >> number;
11 cout << "\n He played knick-knack on ";
12 if(number == 1) //Write out knick-knack information.
13 {
14 cout << "his thumb";
15 }
16 else if(number == 2)
17 {
18 cout << "my shoe";
19 else
20 {
21 cout << "\n whoa! He doesn't play knick-knack there!";
22 }
23 return 0;
24 }
```

Compiler errors

`knickknack.cpp(19) : error C2181: illegal else without matching if
 knickknack.cpp(25) : fatal error C1004: unexpected end of file found`

Always indent code statements within each set of braces. Also, be sure that the opening brace is in the same column as the closing brace. There are several compiler errors that relate to either missing or extra braces. These errors are: *illegal else*, *misplaced else*, *missing brace*, *compound statement missing}* or *unexpected end of file*. If your code will not compile due to one or more of these errors, chances are excellent that you have either too few or too many braces.

Microsoft Visual C++ 2005 Express Tip! To determine the partner of an opening or closing brace, place the cursor on a brace and holding down the control (ctrl) key, press the brace key. The editor will then show which it believes is the partner brace.

Debugging Your Program

If you haven't already, now is the perfect time to learn the debugging tools that accompany your development environment. Refer to Appendix I, "Microsoft Visual C++ 2005 Express Edition Debugger."

Important Note on `cin` and `getline` and Reading Numbers and Strings!

There are three things the C++ programmer needs to keep in mind when working with the `cin` function. It is written so that it ignores leading whitespace characters, such as the enter key and spaces; it reads values until it sees the first space; and it leaves the enter key in the keyboard queue. (It does not remove it.)

The `getline` function that we use to read string data works differently. It is designed to read anything, until it sees an enter key. When `getline` reads data from the keyboard queue, it stops once it sees the Enter key. It removes the Enter key from the queue. If we have a program that reads numbers with `cin` and strings with `getline`, we need to be careful that we don't allow the Enter key, left by `cin`, to be read by `getline`. (This has the effect of `getline` skipping over the `read` statement since it has seen the Enter key, and does not read what you intend.)

Figure 3-11 shows some code and what the user intends to enter into the program. The program asks for a number, and the user enters 42 (Enter key). The `cin` statement leaves that Enter key, which `getline` then sees. Once `getline` sees an Enter key, it believes that it is finished reading from the queue. The user never has a chance to enter the text data.

There is a function that removes the Enter key from the keyboard queue. We must call that function right after we `cin >> number`, so that the Enter key is removed. We must do this if we are reading both string and numeric data. If your program is only reading strings (`getline`), or only reading numbers (`cin`), there isn't a problem. Here is how we use the ignore function to get rid of the Enter key.

```
int number;
string color;
cout << "\n Enter a number: ";
cin >> number;           //user types      42(Enter key)
cin.ignore();            //this removes the Enter key from the queue
cout << "\n Enter a color: ";
getline(cin, name);     //now the getline can read data up to the Enter key
//Remember, getline removes the Enter key, so it isn't problem for the next
//getline, and cin ignores leading Enter keys.
```

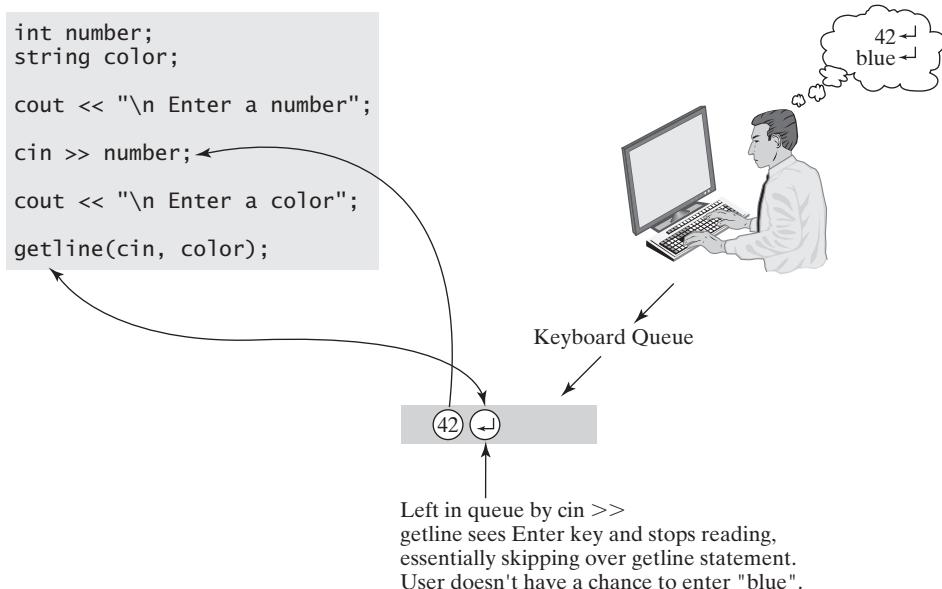


Figure 3-11
Problem with `cin`,
`getline`, and reading
numbers and string data.

3.10

More Fun with C++ classes, the `vector` Class

standard template library (STL)

a large library included in the C++ language that contains many classes for the programmer's use

vector

a container class that is used to hold data elements. The data is held in a linear list.

C++ language has a *standard template library* (referred to as the STL) that is quite large and provides many classes for the C++ programmer. In this chapter, we will learn how to use the `vector` class, which is one of the STL classes. (We shall use several of these classes in this text.) The `vector` class is general purpose, designed to hold data elements in a linear list. When data is added to the `vector`, the `vector` maintains the data in that order. (For those of you who are familiar with arrays, the `vector` is like a dynamic array.) `Vectors` are great to use because they allow programmers to add elements, access them, remove them, and clear the `vector`. Programmers use `vectors` to hold their data because `vectors` can grow and shrink as needed—no need to worry about running out of room (within reason, of course).

Remember that a C++ class is like a job description. The description contains the tasks that are part of doing that job. In C++ we have to make an object of the class, and the object performs various job-tasks. For our purposes we will use a few of the functions that are in the `vector` class. Table 3-5 shows a partial list of the functions and their purpose.

As in all object-oriented programs, we have the class, we make the object and we use the object to call the functions. When making a `vector` object, you must tell the `vector` what type of data it is supposed to hold. Here is a portion of a program where we make two `vectors`, one to hold strings and the other to hold integers. We'll put two names into the `name` `vector` and two numbers into the `numbers` `vector`.

```
#include <vector>           //include this library for vectors
#include <string>            //for string
using namespace std;         //always need this for C++
int main()
{
    vector<int>      vNums;      //make the object named vNums
                                //have to put data type in < >
                                //vNums is a vector of ints
    vector<string>    vNames;    //vNames is a vector of strings

    //now use vNums and vNames objects to call vector functions.
    vNums.push_back(34);        //add 34 and 28 into the numbers vector
    vNums.push_back(28);
    //add Madison and Meggie's names into the name vector
    vNames.push_back("Meggie");
    vNames.push_back("Madison");
```

Program 3-13 demonstrates how we create a `vector` of integers, add values to it using `push_back()`, and the `size()` and `at()` functions to see the number in the `vector` and obtain values from it. The `push_back()` function “pushes” the data onto the end of the `vector`. The `size()` function tells us how many data items are being held in the `vector` object. The `at()` function is passed an integer value and it returns the data item at that element. See the code for more clarification.

TABLE 3-5A Partial List of the C++ STL *<vector>* Class

Function Name	Purpose
<i>at(int)</i>	Given an <i>int</i> , will return the element at that location
<i>push_back()</i>	Adds an element to the end of the vector
<i>pop_back()</i>	Removes the last element in the vector
<i>bool empty()</i>	Return <i>true</i> if the vector is empty, <i>false</i> if it contains at least one element
<i>clear()</i>	Removes all elements from the vector
<i>int size()</i>	Returns the number (integer) of elements in the vector

Program 3-13

```

1 #include <iostream>           //for cout and cin
2 #include <vector>            //for vector
3 #include <iomanip>           //for setw()
4
5 using namespace std;
6
7 int main()
8 {
9     vector<int> vNums;        //vector of ints
10
11    cout << "\n Demonstration of C + + Vectors \n";
12
13    //use the push_back() function to add 5 integers to the vector
14    vNums.push_back(35);
15    vNums.push_back(99);
16    vNums.push_back(27);
17    vNums.push_back(3);
18    vNums.push_back(54);
19
20    //use size() function to see how many numbers are in the vector
21    cout << "\n The vector has " << vNums.size() << " numbers.";
22
23    //add another 2 number
24    vNums.push_back(15);
25    vNums.push_back(72);
26
27    cout << "\n Now there are " << vNums.size() << " numbers in it.";
28
29    //use the at() function to obtain the values from the vector
30    //first element is at(0), second element is at(1)
31
32    cout << "\n Here are the numbers in the vector. \n";
33    for(int i = 0; i < vNums.size(); ++i)

```

```
34      {
35          cout << setw(5) << vNums.at(i);
36      }
37
38      cout << endl;
39      return 0;
40  }
```

Output

Demonstration of C++ Vectors
The vector has 5 numbers.
Now there are 7 numbers in it.
Here are the numbers in the vector.
35 99 27 3 54 15 72.

3.11

Summary

Tables 3-6 and 3-7 summarize the *if else* and *switch* statements as well as the types of loops available in C++. Use these tables as guides for selecting the most appropriate statements for a given task. Appendix G, “Partial C++ Class Reference” provides a summary of the vector class. Turn back to page 582 and look over this reference material.

3.12

Practice!

In these practice programs, we explore how to use the random number generator, the vector and classes, as well as seeing conditional statements and loops in action. We also use code showing how *cin* and *getline* reads data from the keyboard. Study these programs as they will be an excellent resource for you.

Using AND and OR in Conditional Statements

This first program is boring on the outside, but interesting on the inside! It is a simple program that asks the user to enter a whole value. The program then uses two different conditional statements to determine if the value is within the range of -25 to $+25$ (inclusive, meaning -25 and 25 are within the desired range). Often beginning C++ programmers have trouble using the *AND* (*&&*) and *OR* (*||*) operators, and discover problems when trying to make a program work correctly. Figure 3-12 shows a number line with a solid line indicating the desired range in which we are checking and the logical statements that we use. Study the code in Program 3-14 to see how either set of if statements give us the status of the user’s value.

TABLE 3-6*if, if else, if else-if else, switch* Summary

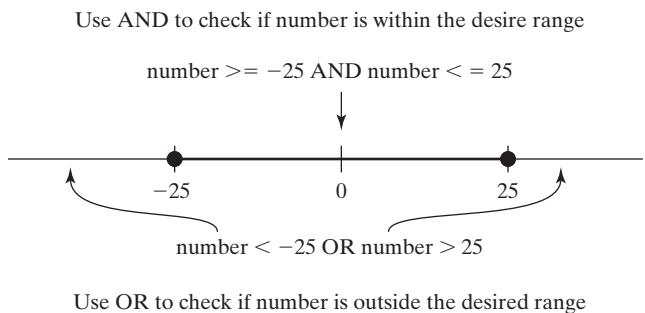
Decision Type	When to Use	Basic Format
<i>if</i>	If a condition is true, perform statements; if the condition is false, skip statements.	<pre>if(condition) { //statement }</pre>
<i>if else</i>	If a condition is true, perform certain statements; if the condition is false, perform different statements.	<pre>if(condition) { // true condition statements } else { // false condition statements }</pre>
<i>if else-if else</i>	If the first condition is true, perform statements and jump to end. If the first condition is false, check the next condition. If it is true, perform statements and jump to end. Continue to check until a condition is true. If no condition is true, perform else statements. Note: else is not required.	<pre>if(condition1) { // true condition1 statements } else if(condition2) { // true condition2 statements } else { // neither condi- tion is true }</pre>
<i>switch</i>	The switch may be used if the value for which you are checking is simple numeric (i.e., 1, 2, 3, etc.) or character (i.e., 'a', 'b', 'c', etc.). Switch will evaluate the expression for a value and performs the case that matches the value. Default statements correspond to else statement in if else structure. Note: Default is not required.	<pre>switch (expression) { case value1: //statements for value 1 break; case value2: // statements for value 2 break; default: // no case matches }</pre>

TABLE 3-7*for, while, and do while Loop Summary*

Loop Type	When to Use	Basic Format for Writing 1 to 10
<i>for</i>	Use the for loop if you know exactly how many times the loop should execute. The condition must be true for it to run.	<pre>int i; for(i = 1; i <= 10; ++i) { cout << "\n i =" << i; }</pre>
<i>while</i>	Use the while loop when a loop must continue to run until a condition has been met. The condition must be true for it to run.	<pre>int i = 1; while(i <= 10) { cout << "\n i =" << i; ++i; }</pre>
<i>do while</i>	Use the do while loop when the loop statements must run at least once. The condition will be checked after the first pass, and the loop will continue as long as the condition is true.	<pre>int i = 1; do { cout << "\n i =" << i; ++i; } while(i <= 10);</pre>

Figure 3-12

AND and OR operators are used for determining if a value falls inside or outside a given range.

**Program 3-14**

```

1 //This program demonstrates the AND Operator
2 //and OR operator for checking values inside and outside
3 //of numeric ranges.
4
5 //NOTE: if this were a "real" program, you could use
6 // just the AND or just the OR, no need to use both!
7
8 #include <iostream>
9 #include <string>
10 using namespace std;
11
```

```
12 int main()
13 {
14     int number;
15     string answer;
16
17     cout << "\n This program asks the user to enter a whole number."
18         << "\n It then determines if the number is between -25 to + 25"
19         << "\n using both AND and OR operators.";
20
21     do
22     {
23         cout << "\n\n Please enter a whole number.  ";
24         cin >> number;
25         cin.ignore();           //strip out Enter key left from cin
26
27         //One way to check, use OR operator, the number
28         //could be below low range OR above high range.
29
30         if(number < -25 || number > 25)
31         {
32             cout << "\n The number " << number << "is outside
33             +/-25.";
34         }
35         else
36         {
37             cout << "\n The number " << number << "is inside
38             +/-25.";
39         }
40
41
42         //A different way to check, use AND operator, the number
43         //could be above low value AND below high value.
44
45         if(number > = -25 && number < = 25)
46         {
47             cout << "\n The number " << number << "is inside
48             +/-25.";
49         }
50         else
51         {
52             cout << "\n The number " << number << "is outside
53             +/-25.";
54         }
55
56         cout << "\n See another number?  yes/no ";
57         getline(cin,answer);
58
59     }while(answer  == "yes");
60
61     return 0;
62 }
```

Output

This program asks the user to enter a whole number.
It then determines if the number is between -25 to + 25
using both AND and OR operators.

```
Please enter a whole number. 4
The number 4 is inside +/-25.
The number 4 is inside +/-25.
See another number? yes/no yes
```

```
Please enter a whole number. -26
The number -26 is outside +/-25.
The number -26 is outside +/-25.
See another number? yes/no no
```

PI Estimation Using an Infinite Series Calculation

In mathematics, an infinite series is represented by a sum or sequence of terms. Infinite series typically have a formula that can be used to calculate an estimate of a certain value. Below is an infinite series approximation to calculate an estimated value of PI:

$$pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} - \frac{4}{15} + \frac{4}{17} \dots$$

In theory, the more terms you use, the closer to PI you get. (The above equation is showing 9 terms.) For example, if you use 5 terms to calculate pi, your formula would yield the following result:

$$\begin{aligned} pi &= 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} = 4 - 1.333333333333 + 0.8 - 0.571428 \\ &\quad + 0.4444444 = 3.33968254 \end{aligned}$$

In this program, we ask the user for the number of terms in which to calculate PI, then use a *for* loop for the calculation loop. In each pass of the loop, we calculate another term and add it to the sum that represents our calculated value of PI. The trick in this program is making the sign of the term flip from positive to negative. Multiplying the numerator value (4.0) by -1.0 flips the sign for us. One last thing, notice how we have to initialize the numerator and denominator values above the *for* loop—this is because if we perform more than one calculation, we need to be sure to reset these values! (Don't believe me? Move the initialization statements outside the *do while* loop and see the goofy results we get on the second PI calculation!).

Program 3-15

```
1 //This program uses an infinite series to
2 //calculate an estimated value of PI.
3
4 //Here is PI to 20 places
5 //3.14159265358979323846
6
7 #include <iostream>
8 #include <string>
9 #include <iomanip>           //for setprecision
10 using namespace std;
11
12 int main()
13 {
14     int nTerms;
15
16     //doubles can only store 15 places
17     double PI = 3.141592653589793;
18
19
20     double numerator, denom;
21     double piCalc;
22     string answer;
23
24     cout << "\n PI Calculation Program\n";
25
26     do
27     {
28         cout << "\n Enter number of terms for PI calculation: ";
29         cin >> nTerms;
30         cin.ignore();    //strip out Enter key left over from cin
31
32         //Since we are summing up PI, zero this variable!
33         piCalc = 0.0;
34
35         //Set the values for these variables that change.
36         numerator = 4.0; //start at +4.0
37         denom = 1.0;      //start at 1.0
38         for(int i = 0; i < nTerms; ++i)
39         {
40             piCalc = piCalc + numerator/denom;
41
42             denom += 2.0;
43             numerator = -1.0 * numerator; //flip sign
44         }
45
46         cout << "\n Results\n Number of Terms: " << nTerms
47             << setprecision(10) << "    PI = " << PI
```

```

48             << "    Calc PI    = " << piCalc << endl;
49
50         cout << "\n Do more PI calculations? yes/no ";
51         getline(cin,answer);
52
53     }while(answer == "yes");
54
55     return 0;
56 }

```

Output

PI Calculation Program

Enter number of terms for PI calculation: 5

Results

Number of Terms: 5 PI = 3.141592654 Calc PI = 3.33968254

Do more PI calculations? yes/no yes

Enter number of terms for PI calculation: 500000

Results

Number of Terms: 500000 PI = 3.141592654 Calc PI = 3.141590654

Do more PI calculations? yes/no yes

Enter number of terms for PI calculation: 5000000

Results

Number of Terms: 5000000 PI = 3.141592654 Calc PI = 3.141592634

Do more PI calculations? yes/no no

Random Number Generator Part 1: *rand()* and *srand()*

rand()

in the standard namespace, a function that generates random numbers between 0 – 32767

srand()

in the *cstdlib* library, the seed function for random number generator

The standard namespace contains many utility functions for the C++ programmer. (We have access to this when we place “using namespace std;” at the top of our *main* function.) It contains a random number generator named *rand()*, as well as its seed function, *srand()*. The *rand()* function is a pseudo-random number generator that returns an integer between 0 – MAX_RAND (usually, 32,767). The *srand()* function is passed an integer value, the seed, which sets the starting point for generating random numbers. If you use the same seed value, you obtain the same series of numbers from *rand()*. Examine Program 3-16. It uses the time from the computer to seed the generator, and then obtains seed values from the user. Notice that if we use the same seed value (123), we obtain the same series of numbers.

Program 3-16

```

1 //Demonstrate how rand() and srand() operate.
2
3
4 #include <iostream>                      //for cout, cin
5 #include <iomanip>                       //for setw()
6 #include <string>                         //for string

```

```
7 #include <ctime>           //for time()
8
9 using namespace std;
10
11 int main()
12 {
13     cout << "\n Random Number Generator Demo ";
14
15     int seed, number, i;
16     string answer;
17
18     cout << "\n\n First, here are 8 random numbers."
19         << "\n We used the time function for seed value.\n";
20
21     //seed with time, and see what we get
22     srand( (unsigned)time( NULL ) );
23     for(i = 0; i < 8; ++i)
24     {
25         number = rand();
26         cout << setw(8) << number;
27     }
28
29     cout << "\n Now we'll let you choose the seed value. ";
30     do
31     {
32
33         cout << "\n\n Please enter a seed value for rand() ";
34         cin >> seed;
35
36         //cin >> leaves the enter key in keyboard queue
37         //we need to strip it off so getline doesn't read it
38         cin.ignore();
39
40         srand(seed);
41         cout << "\n Eight numbers with seed " << seed << endl;
42
43         for(i = 0; i < 8; ++i)
44         {
45             number = rand();
46             cout << setw(8) << number;
47         }
48
49         cout << "\n Do another set of numbers? yes/no ";
50         getline(cin,answer);
51
52     }while(answer == "yes");
53
54     return 0;
55 }
```

Output

```
Random Number Generator Demo
```

First, here are 8 random numbers.

We used the time function for seed value.

```
22657 7256 5711 19109 12007 1668 6644 23166
```

Now we'll let you choose the seed value.

Please enter a seed value for rand() 123

Eight numbers with seed 123

```
440 19053 23075 13104 32363 3265 30749 32678
```

Do another set of numbers? yes/no yes

Please enter a seed value for rand() 5

Eight numbers with seed 5

```
54 28693 12255 24449 27660 31430 23927 17649
```

Do another set of numbers? yes/no yes

Please enter a seed value for rand() 123

Eight numbers with seed 123

```
440 19053 23075 13104 32363 3265 30749 32678
```

Do another set of numbers? yes/no no

Random Number Generation Part 2: Use *mod* % Operator

Often programmers need to generate random numbers within a certain range of values. If we write a program that needs to select a number at random, say between 1 and 10, the *rand()* function could be used, but as it stands now, what are the chances of getting a value between 1 and 10 when it generates numbers as high as 32,767? If we need a range of values from -25 to +25, or between 0.000 and 1.000, how can we use *rand()* if it only generates values between 0 and 32767?

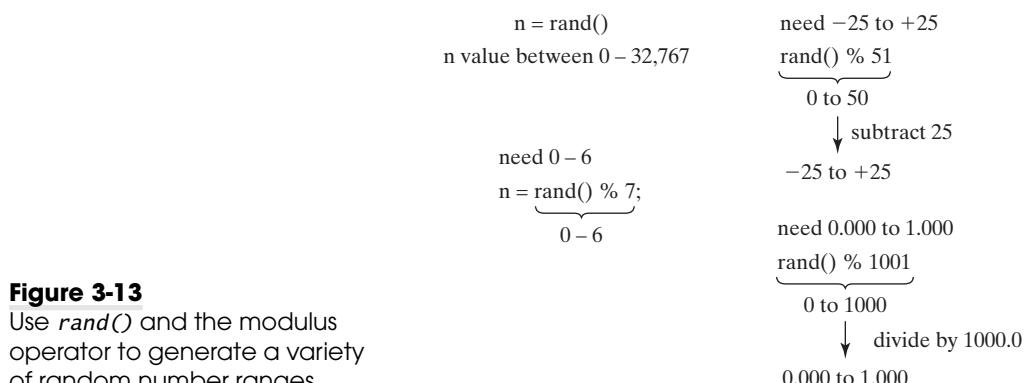


Figure 3-13

Use *rand()* and the modulus operator to generate a variety of random number ranges.

The modulus operator (and a bit of arithmetic) comes to our rescue for generating random numbers within certain ranges. Remember that the % operator gives you the whole number remainder in integer division. (Review Chapter 2, Figure 2-8.) We use this principle to adjust the values from *rand()*, then use simple arithmetic to obtain our desired range of values. Figure 3-13 shows how the modulus operator combined with arithmetic gives us three sets of random numbers within a given range. Program 3-17 shows the code for producing ten values for each of the three ranges of random numbers described above.

Program 3-17

```
1 //A demonstration of how you can obtain the
2 //desired range of random numbers using % and arithmetic.
3
4
5 #include <iostream>           //for cout and cin
6 #include <iomanip>           //for setw()
7 #include <ctime>              //for time()
8 using namespace std;
9
10 int main()
11 {
12
13     cout << "\n This program generates three sets of random"
14         << " numbers \n using the % operator and arithmetic\n";
15
16     //Use the time to seed rand()
17     srand( (unsigned)time(NULL) );
18
19     int number;
20
21     cout << "\n If we rand()%N, we get values between 0 and N-1 "
22         << "\n Here we pick N of 7, we get values between
23         0 - 6.\n";
24
25     for(int i = 0; i < 10; ++i)
26     {
27         number = rand()%7;           //gives us values from 0 - 6
28         cout << setw(5) << number;
29     }
30
31
32     cout << "\n\n For values between 0.000 and 1.000, we generate "
33         << "\n integers between 0 - 1000, and divide by 1000. \n";
34
35     double random;
36     cout.precision(3);
37     cout.setf(ios::fixed);
```

```

38         for(i = 0; i < 10; ++i)
39         {
40             number = rand()%1001; //gives us values from 0 to 1000
41             random = number/1000.0; //divide by 1000.0 not 1000
42             cout << setw(7) << random;
43         }
44
45
46         cout << "\n\n For a value between -25 and 25, obtain 0 - 50"
47             << "\n and subtract 25. \n";
48         for(i = 0; i < 10; ++i)
49         {
50             number = rand()%51; //gives us 0 to 50
51             number = number - 25;
52             cout << setw(5) << number;
53         }
54
55         cout <<"\n\n Pretty cool, isn't it? " << endl;
56
57         return 0;
58     }

```

Output

This program generates three sets of random numbers using the % operator and arithmetic

If we `rand()%N`, we get values between 0 and N-1

Here we pick N of 7, we get values between 0 – 6.

3 4 2 0 4 6 1 3 5 0

For values between 0.000 and 1.000, we generate integers between 0–1000, and divide by 1000.

0.866 0.266 0.040 0.312 0.366 0.944 0.964 0.577 0.706 0.467

For a value between -25 and 25, obtain 0 – 50 and subtract 25.

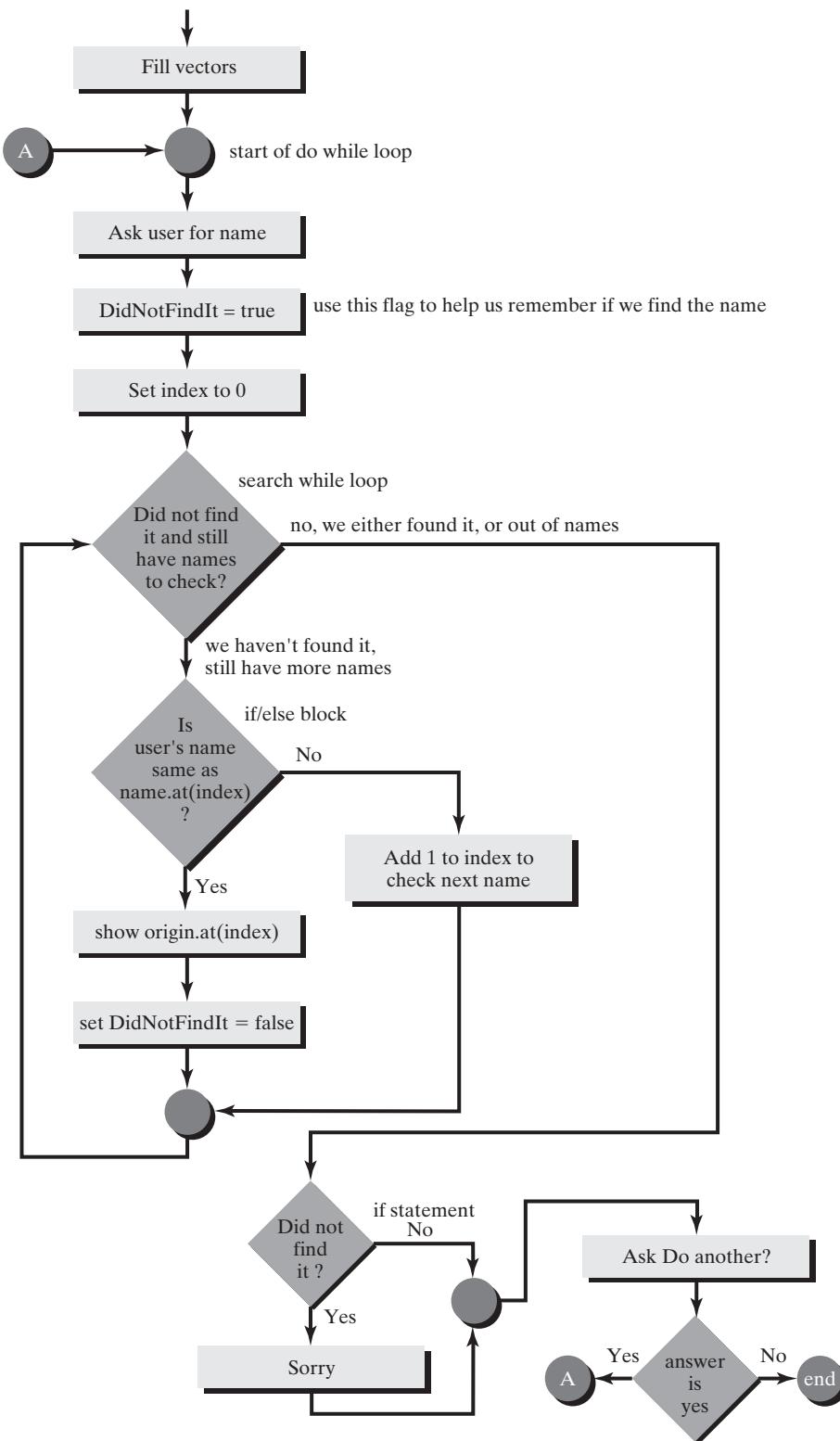
20 22 -23 16 -25 13 0 -17 24 16

Pretty cool, isn't it?.

Search for the Name and Program Flags

This program uses everything we've learned in this chapter! We are going to build a program that uses `if` statements, loops, vectors, and strings. In this program, we allow the user to enter a woman's name, and we search our vector of names to see if we have it. If we do find the name, we report the origin and meaning of the name. If we do not find the name, we tell the user we are sorry but do not have the name in our vector.

Figure 3-14 shows a flow diagram for this program's logic. Program 3-18 shows the C++ complete program. In this code we need to know (or remember) if

**Figure 3-14**

Flow diagram for the Search for the Name program. The `bDidNotFindIt` variable helps to know if the program has found the woman's name in the vector. This variable is known as a program flag, as it "flags" or identified a condition or state in the program.

program flag or flag variable

a program variable (usually a bool or an int) that is set either true/false or given a value to help the program remember states or conditions

we have found the woman's name in our vector so we can stop searching. Our program has a boolean variable named "bDidNotFindIt," which we set to true before we start searching. (That is, we did not find it yet.) If we find the name, we set bDidNotFindIt to false, that is, no, we did find it! When we use a variable like this where we set it to true or false depending on the condition in the program, that variable is referred to as a **program flag** or a **flag variable**.

Program 3-18

```
1 //Program asks the user to enter a woman's name.
2 //Search our vector to see if the name is in it.
3 //If it is, write out the origin/meaning of the name.
4
5 #include <iostream>           //cout, cin, getline
6 #include <string>             //string object
7 #include <vector>              //vector objects
8
9 using namespace std;
10
11 int main()
12 {
13
14     cout << "\n Welcome to the C + + Name Search Program";
15
16     //Create 2 string vectors, names and origins.
17     vector<string> names;
18     vector<string> origins;
19
20     //First load up our vectors.
21     names.push_back("Barbara");
22     origins.push_back("greek, meaning stranger");
23
24     names.push_back("Kelly");
25     origins.push_back("celtic, meaning warrior or defender");
26
27     names.push_back("Claire");
28     origins.push_back("french, meaning bright and clear");
29
30     names.push_back("Janis");
31     origins.push_back("english, God is gracious");
32
33     names.push_back("Ciara");
34     origins.push_back("celtic, meaning black and mysterious");
35
36     names.push_back("Lucy");
37     origins.push_back("latin, meaning bringer of light");
38
39     //Variables for the guessing logic
40     string answer, userName;
```

```
41
42     do
43     {
44         bool bDidNotFindIt = true;
45
46         cout << "\n Please enter a woman's name, such as Kelly: ";
47         getline(cin,userName);
48
49         //Search our name vector until we either find it, or
50         //run out of names. Stop looking when we find it.
51
52         int index = 0;
53         while(index < names.size()  && bDidNotFindIt == true)
54         {
55             if(userName == names.at(index))
56             {
57                 cout << "\n Here is your name: "
58                 << names.at(index);
59                 cout << "\n Origin: " << origins.at(index);
60
61                 bDidNotFindIt = false;      //we did find it!
62             }
63             else
64             {
65                 ++ index;
66             }
67         }
68
69         //check the bDidNotFindIt variable
70         //if it is still true, we didn't find the user's name
71         if(bDidNotFindIt)
72         {
73             cout << "\n Sorry. " << userName << " isn't in our
74             vector. ";
75         }
76
77         cout << "\n\n Do another name? yes/no ";
78         getline(cin,answer);
79         cin.ignore();
80
81     }while (answer == "yes");
82
83     return 0;
84 }
```

Output

Welcome to the C++ Name Search Program

Please enter a woman's name, such as Kelly: Ciara
Here is your name: Ciara

```
Origin: celtic, meaning black and mysterious
Do another name? yes/no yes
Please enter a woman's name, such as Kelly: Elizabeth
Sorry. Elizabeth isn't in our vector.

Do another name? yes/no yes
Please enter a woman's name, such as Kelly: Kelly
Here is your name: Kelly
Origin: celtic, meaning warrior or defender

Do another name? yes/no no
```

Time Conversion

This program is an elapsed time-conversion program that presents the user with three choices:

1. Convert time (in hours, minutes, and seconds format) to total seconds.
2. Convert total seconds to hours, minutes, and seconds.
3. Exit the program.

The program presents the user with a numeric menu and uses a *switch* statement to handle the user's input. This "old style" menu system was popular in the old-days of programming, before graphics cards and mice were available. In today's programming environment, our end-users expect a graphical user interface with on-board help, pretty icons, and input error checking!

There are two things to notice here in Program 3-19. First when we read the H:M:S value from the keyboard, we do not read it as a string, but read using *cin*. Here we make use of *cin*'s ability to read each of the five pieces of the input data individually (one integer, a character, an *int*, a *char*, and another integer). There is a *char* variable named "colon" that is used to read the ":" from the input queue. The second thing to note here is that we have a *do while* loop set up so that we continue looping until the user enters a 3. This avoids the "do another" question structure we have in the previous programs.

Program 3-19

```
1 //Time Conversion Demonstration
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     int choice,hr, min, sec, totalsec;
9     char colon;
10
11    cout << "\n A program that converts between H:M:S and seconds. ";
12
13    do
```

```
14     {
15         cout << "\n\n 1 = Convert H:M:S to seconds"
16             << "\n 2 = Convert seconds to H:M:S"
17             << "\n 3 = Exit"
18             << "\n\n Please pick your choice ==> ";
19         cin >>choice;
20
21     switch(choice)
22     {
23         case 1:           //convert H:M:S to seconds
24
25             cout << "\n Enter H:M:S format, i.e. 3:26:33 ==> ";
26             cin >> hr >> colon >> min >> colon >> sec;
27             totalsec = hr*3600 + min*60 + sec;
28             cout << " Result: " << totalsec;
29             break;
30
31         case 2:           //convert seconds to H:M:S
32
33             cout << "\n Enter seconds, i.e. 3440 ==> ";
34             cin >> totalsec;
35             hr = totalsec/3600;
36             totalsec = totalsec - hr*3600;
37             min = totalsec/60;
38             totalsec = totalsec - min*60;
39             sec = totalsec;
40             cout << " Result: " << hr << ":" << min << ":" << sec;
41             break;
42
43         case 3:           //exiting program
44
45             cout << "\n You have chosen to exit." << endl;
46             break;
47
48         default:
49             cout << "\n Oh I don't do that choice! Try again! ";
50     }
51
52 }while(choice != 3);
53
54 return 0;
55 }
```

Output

A program that converts between H:M:S and seconds.

```
1 = Convert H:M:S to seconds
2 = Convert seconds to H:M:S
3 = Exit
Please pick your choice ==> 1
```

```

Enter H:M:S format, i.e. 3:26:33 ==> 4:38:57
Result: 16737

1 = Convert H:M:S to seconds
2 = Convert seconds to H:M:S
3 = Exit
Please pick your choice ==> 2

Enter seconds, i.e. 3440 ==> 16737
Result: 4:38:57

1 = Convert H:M:S to seconds
2 = Convert seconds to H:M:S
3 = Exit
Please pick your choice ==> 3
You have chosen to exit.

```

Vowel Counting Program

Let's revisit the string class, and write a simple program that asks the user to enter a sentence. In Program 3-20 we'll read the sentence into a string, and then using the string's `at()` function, pull out a character at a time, counting the sentence for its vowels (a, e, i, o, u, A, E, I, O, and U). Instead of checking for lower-and uppercase letters, when we obtain the letter from the string, we'll convert it to lowercase using the ctype's `tolower()` function. The `ispunct()` and `isspace()` functions help count the spaces and punctuation marks in the sentence. For variety's sake, we'll use a `while` loop instead of a `do while` loop. Notice how we have to initialize the answer to "yes" before we enter the user's input loop.

Program 3-20

```

1 //Program to search a string for the 5 vowels,
2 //and count the punctuation marks and spaces.
3
4 #include <iostream>           //for cout, getline
5 #include <string>             //for string
6 #include <cctype>              //for tolower(), ispunct(), isspace()
7
8 using namespace std;
9
10 int main()
11 {
12
13     cout << "\n Welcome to the C++ Vowel Counter Program";
14
15     string answer = "yes"; //need to initialize
16
17     string sentence;
18     char letter;
19

```

```
20     int aCount, eCount, iCount, oCount, uCount;
21     int punctCount, spaceCount;
22
23     while(answer == "yes")
24     {
25         //Set the count values to zero,
26         //so we get an accurate count with each sentence.
27         aCount = eCount = iCount = oCount = uCount = 0;
28         punctCount = 0;
29         spaceCount = 0;
30
31         cout << "\n Enter a sentence \n => ";
32         getline(cin,sentence);
33
34         for(int i = 0; i < sentence.size(); ++i)
35         {
36             letter = sentence.at(i);
37             letter = tolower(letter); //convert letter here
38
39             switch(letter)
40             {
41                 case 'a':
42                     aCount++;
43                     break;
44                 case 'e':
45                     eCount++;
46                     break;
47                 case 'i':
48                     iCount++;
49                     break;
50                 case 'o':
51                     oCount++;
52                     break;
53                 case 'u':
54                     uCount++;
55                     break;
56             }
57             if(ispunct(letter) != 0) //return non-zero if true
58                 punctCount++;
59
60             if(isspace(letter) > 0 ) //returns non-zero if true
61                 spaceCount++;
62         }
63
64         cout << "Results from: " << sentence;
65         cout << "\n A/a " << aCount
66             << "\n E/e " << eCount
67             << "\n I/i " << iCount
68             << "\n O/o " << oCount
```

```

69             << "\n U/u " << uCount
70             << "\n Punctuation marks " << punctCount
71             << "\n Spaces " << spaceCount << endl;
72
73     cout << "\n Do another sentence? yes/no ";
74     getline(cin,answer);
75 }
76 return 0;
77 }
```

Output with answering yes once, then no

Welcome to the C++ Vowel Counter Program

Enter a sentence

=> Alice, the panda, enters Ed's Bar and eats shoots and leaves.

Results from: Alice, the panda, enters Ed's Bar and eats shoots and leaves.

A/a 8

E/e 8

I/i 1

O/o 2

U/u 0

Punctuation marks 4

Spaces 10

Do another sentence? yes/no yes

Enter a sentence

=> Oscar enters Uptown Bar and eats, shoots, and leaves.

Results from: Oscar enters Uptown Bar and eats, shoots, and leaves.

A/a 6

E/e 5

I/i 0

O/o 4

U/u 1

Punctuation marks 3

Spaces 8

Do another sentence? yes/no no

Pond Pump Calculator Program

It's time to roll up our sleeves and get our hands wet as we write a program that helps determine the correct pond pump for your backyard water feature. When you design your pond, it is important that all of its water is cycled through the filter twice, in one hour. This means that if you build your pond so that it holds 500 gallons, the pump should be sized to be at least a 1000 gallons per hour. The C++ Pond Pump Manufacturers build excellent pond pumps guaranteed for five years and they all run on two AA batteries.

This program is designed to calculate the correct pump size for a rectangular pond. The program has a vector that contains the eight pumps available to the pond

builder. The user enters the rectangular pond dimensions and the program calculates the total capacity and doubles it to determine the pump size. In the case where the pond is very small, or very large, the program sets a boolean “*bGoodFit*” flag to false, indicating that perhaps the recommended C++ pump is not ideal for the user’s pond. Read through the program comments in Program 3-21 to gain further insight into this program. Remember, too, that there are 231 cubic inches of water in a gallon of water.

Program 3-21

```
1 //This program ask the user to enter the dimensions
2 //of her rectangular pond. It then calculates
3 //the volume and required pump size.
4
5 #include <iostream>
6 #include <vector>
7 #include <string>
8
9 using namespace std;
10
11 int main()
12 {
13     cout << "\n C++'s Pond Pump Calculator Program"
14         << "\n We'll determine the best C++ pump"
15         << " for your rectangular pond.";
16
17     cout << "\n\n Your pump should be sized so that the entire"
18         << " pond volume \n pumps through the filter twice"
19         << " in one hour.";
20
21     vector<int> pumps;      //hold various pump sizes
22
23     string answer;
24
25 //First fill the pumps vector with the possible pumps
26 //based on their gallons per hour(gph) pump rate.
27
28 //C++ pumps come in these gph sizes:
29 //350, 600, 950, 1200, 1600, 2500, 3200, 5000 gph
30     pumps.push_back(350);
31     pumps.push_back(600);
32     pumps.push_back(950);
33     pumps.push_back(1200);
34     pumps.push_back(1600);
35     pumps.push_back(2500);
36     pumps.push_back(3200);
37     pumps.push_back(5000);
38
39 //we'll assume pond dimensions are in inches
40     int pondWidth, pondDepth, pondLength;
```

```
41
42     do
43     {
44         cout << "\n\n Enter the pond's length, width, and depth: ";
45         cin >> pondLength >> pondWidth >> pondDepth;
46         cin.ignore(); //remove enter key getline is next read
47
48         int cubicInchVol = pondLength * pondWidth * pondDepth;
49         float totalGallons = cubicInchVol/231.0;
50
51         cout << "\n Your pond is " << totalGallons << " gallons.";
52
53         //determine best sized pump
54         //figure out what pump needs to pump per hour
55         int twiceCap = totalGallons * 2.0;
56
57         int numPumps = pumps.size();
58         int neededPumpIndex;
59         bool bGoodFit = true; //set false if pump not adequate
60
61         //First, is it a small pond?
62         if(twiceCap < pumps.at(0))
63         {
64             neededPumpIndex = 0;
65             if(twiceCap < pumps.at(0)/3) //if pump is 3x too big
66             {
67                 bGoodFit = false;
68             }
69         }
70         //Or is it a very large pond?
71         else if(twiceCap > pumps.at(numPumps-1) )
72         {
73             neededPumpIndex = numPumps-1;
74             bGoodFit = false;
75         }
76         else //our pond needs are somewhere within range of pumps
77         {
78             for(int i = 0; i < numPumps-1; ++i)
79             {
80                 if( twiceCap > pumps.at(i) && twiceCap <
81                     pumps.at(i+1) )
82                 {
83                     //found our range, recommend larger pump
84                     neededPumpIndex = i+1;
85
86                     break; //can now break out of for loop
87                 }
88             }
89         }
90     }
```

```
89
90     if(bGoodFit == true)
91     {
92         cout << "\n Your pond needs the "
93         << pumps.at(neededPumpIndex) << " pump." << endl;
94     }
95     else
96     {
97         cout << "\n The " << pumps.at(neededPumpIndex)
98         << " pump is the only one that we have for your pond. "
99         << "\n There may be a better pump for you. " << endl;
100    }
101
102    cout << "\n Figure another pond pump? yes/no ";
103    getline(cin, answer);
104
105 }while(answer == "yes");
106 return 0;
107 }
```

Output

C++'s Pond Pump Calculator Program

We'll determine the best C++ pump for your rectangular pond.

Your pump should be sized so that the entire pond volume
pumps through the filter twice in one hour.

Enter the pond's length, width, and depth: 40 25 12

Your pond is 51.9481 gallons.

The 350 pump is the only one that we have for your pond.

There may be a better pump out there for you.

Figure another pond pump? yes/no yes

Enter the pond's length, width, and depth: 84 50 30

Your pond is 545.455 gallons.

Your pond needs the 1200 pump.

Figure another pond pump? yes/no yes

Enter the pond's length, width, and depth: 120 72 34

Your pond is 1271.69 gallons.

Your pond needs the 3200 pump.

Figure another pond pump? yes/no no

REVIEW QUESTIONS AND PROBLEMS**Short Answer**

1. Describe the precedence of operations for the relational and logical operators.
2. What is the difference between a unary and a binary operator?

3. What is the best technique for checking whether a floating point or double value is the same as exactly zero?
4. When are braces required in an *if* statement?
5. Is it possible to nest a *switch* statement inside an *if* statement?
6. What are the three different methods for performing a loop in C++?
7. Why is a loop altering statement necessary in a *while* loop?
8. What type of loop always performs the loop statements at least once?
9. What is (are) the consequence(s) if you forget to break your switch?
10. Name the four keywords associated with a *switch* statement.
11. What is the difference between the *push_back()* and *pop_back()* vector class functions?
12. What is the difference between using a *break* in a *for* loop and using a *continue* in a *for* loop?
13. If you wrote a program asking the user to enter a color (such as red, blue, etc.) would it be possible to use a *switch* statement to compare the user's color to a program color? (In other words, could you use as a case statement like this? `case "red":`)
14. Is it possible to store a list of strings and numbers in the same vector? Explain your answer.
15. If you have a program that may or may not execute a *while* loop, is it possible to replace that *while* loop with a *do while* loop? Explain your answer.
16. Why are Boolean variables useful in *if* statements and loops?
17. The random number generator function *rand()* returns what sort of value to the programmer? What is the minimum and maximum possible values from *rand()*?
18. Explain how the *for* loop is executed. Does the increment step happen before or after the first pass of the loop?
19. Is it possible to write a *for* loop that does not execute when the program runs? A *while* loop? A *do while* loop?
20. Compare how the *if* statement and the *switch* statement are alike (site 3 ways). Explain how they're not alike (site 2 ways).

Debugging Problems: Compiler Errors

Identify the compiler errors in Problems 21 to 24 and state what is wrong with the code.

21.

```
int a = 7, b = 9, c = 2;
If(a << b)
{
    c == b;
}
```

22.

```
int 3_for_me = 3, quick_4
switch(3_for_me)
{
    Case 7: cout << "hello"; break;
    case 8: cout << "goodbye"; break;
}
```

23.

```
float inventory, case;
inventory = 8;
if(inventory = 3)
{
    case = inventory;
    inventory = 0;
}
```

24.

```
int Hurry = 17, m, n, o;
if(m < n)
    o = Hurry;
    n = o;
else
m = Hurry;
```

Debugging Problems: Run-Time Errors

Each of the programs in Problems 25 to 27 compiles but does not do what the specification states. What is the incorrect action and why does it occur?

25. Specification: Write out Hello World twenty-five times. Each hello begins a new line.

```
#include <iostream>
using namespace std;
int main()
{
    int i = 1;
    while(i < 25)
        cout << "\nHello World";
        ++i;
    return 0;
}
```

26. Specification: Check to see if the user's input is a 0 or a 1. If it is a 0 or a 1, write out Hello World.

```
#include <iostream>
using namespace std;
int main()
{
    int user_input;
```

```

        cout << "\nEnter an integer.";
        cin >> user_input;
        if(user_input == 0 || 1)cout << "\nHello World";
        return 0;
    }
}

```

- 27.** Specification: Check to see if the user's input is 1, 2, or 3. Write out the numeric word (such as ONE) if it is within range; otherwise, write OUT OF RANGE.

```

#include <iostream>
using namespace std;
int main()
{
    int user_input;
    cout << "\nEnter an integer.";
    cin >> user_input;
    switch(user_input)
    {
        case 1:
        cout << "\nONE";
        case 2:
        cout << "\nTWO";
        case 3:
        cout << "\nTHREE";
        default: cout << "OUT OF RANGE";
    }
    return 0;
}

```

Reading the Code

In Problems 28–31, what will be the output from the source code?

28.

C++ Code	Output
<pre> int x = 10, y = 6, ctr = 0; cout << "Hi There!" << endl << endl; while(ctr < 3) { cout << x << y; ++ctr; y--; } do { cout << "\nI Love C++!"; }while(ctr < 3); </pre>	

29.

C++ Code	Output
<pre>int a = 5, b = 8; while(a < b) { cout << "\n Red"; ++a; } do { cout << "\n Blue"; ++a; }while(a < 6); cout << "\n END!";</pre>	

30.

C++ Code	Output
<pre>int i, j; for(i = 0; i < 5; ++i) { cout << "\n Hi"; if(i == 1) break; } for(j = 0; j < 5; ++j) { cout << "\n Hey you."; if(j%2 == 0) continue; cout << "\n How's things?"; }</pre>	

31.

C++ Code	Output
<pre>int a=4, b=10, sum; bool bGo = true; while(bGo == true) { cout << "\n a= " << a; cout << " b = " << b; ++a; b = b-2; sum = a+b; if(sum == 12) bGo = false; }</pre>	

Writing Loop Code

In Problems 32–35, write the same code using the designed loop structure. You are not writing complete programs!

32. Write a *for* loop and a *do while* loop that writes the numbers 13 to 24. The list of numbers should be on one line in the output window, separated by a comma, like this: 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24.
33. Declare a string variable and ask the user to enter a line of text. This process should continue until the user enters the word “done”. Write a *while* loop and a *do while* loop that accomplishes this task.
34. Ask the user to enter her full name. (Read into a string variable.) Determine the number of characters in her name and then write the name to the screen that many times. Do not count the spaces. “Mary Jane” will result in her name being written 8 times. Illustrate this task using both a *for* loop and *while* loop.
35. Write a *for* loop and a *do while* loop that writes integers to the screen in descending order, from 20 to 1. Write the numbers as two lines.

Challenge Code

Once in a while C++ programmer encounter “intelligence tests” that have the programmer examine some source code and make corrections or modifications to it. Here are two challenges for you to try.

36. The code below is to write a dash “-” to the screen 40 times. You can replace, remove, or add one character in the code. Can you find 2 solutions? (What does the current code do?)

```
int j, k = 40;
for( j = 0; j < k; j)
    cout << "-";
```

37. The code below needs to write “hello world” the screen eight (8) times. You can replace, remove, or add one character in the code. Can you find 2 solutions? (What does the current code do?)

```
int c = 0, t = 9;
while( c < t)
{
    cout << "\n hello world";
    ++c;
}
```

Programming Problems

For all of the following programming problems, have the program write your name and program title to the screen one time. Incorporate a “play again” loop so that the user can repeat the program. Have the program ask if the user wishes to go again utilizing a string for the “yes” or “no” response. When the user is finished “playing,” have the program write out a goodbye message to the screen. Note: do not “trap” the user in a loop forcing him to enter “valid” data! For these problems, if you obtain invalid data, write an error and drop to the “go again” code.

38. Write a complete C++ program that asks the user for a number between 0 and 100 (0 and 100 are out of range). If the number is between 1 and 9, write out the words ONE DIGIT BIG! If it is between 10 and 99, write out the words TWO DIGITS BIG! If the user's number is outside the requested range write the phrase OUT OF RANGE. Your program should use an *if-else/if-else* control structure, not three individual *if* statements.
39. Write a complete C++ program that asks the user to enter a date in the month/day/year format, such as 4/25/2007. Check to ensure that the date is valid. (Remember, thirty days hath September, April, June, and November. All the rest have thirty-one, except in leap year, once in four, when February has one day more.)¹ If the date is valid, state that, as well as convert it to the day number in the year. For example, January 31 is the thirty-first day of the year, and February 1 is the thirty-second day of the year. Determine if the year is a leap year. Recall that a year is a leap year if it is evenly divisible by 4—except in century years, which are leap years only if they are divisible by 400. If the date is not valid, write a “not valid” message.
40. Write a complete C++ program that asks the user to input a character. Using the ASCII character set as a guide, state whether the user's character is a digit (0 to 9), a letter (a to z or A to Z) or a symbol. Include the character's decimal, octal, and hex value as well in your output.
41. Write a complete C++ program that converts distance values. The program should give the user three options: convert a whole number of inches to feet and inches, convert feet and inches to decimal feet or feet and inches to inches. For example, 80 inches is 6 feet, 8 inches; and 5 feet, 6 inches is 5.5 feet and 4 feet, 2 inches is 50 inches. Write the decimal feet to three decimal places.
42. Write a complete C++ program that prints out one of the output patterns shown below. (You should select only one pattern to program. You are not being asked to build all three.)

```
+           +           +   +
++          +++          +   +
+++         +++++        +   +
++++        +++++++      +
+++++       ++++++++
+++++
++++
+++
+
```

Ask the user for a symbol and the number of lines (limit 1 to 20). If the user enters an invalid number of lines, issue an error message. The program should then ask the user if he would like to go again. If the answer is yes, have the program loop back to the point in the code where it asks for the lines and the symbol.

¹A common saying dating back perhaps to Holinshed's Chronicals of England. There have been many modifications since 1577.

43. Write a C++ program named Guess the Color. This program gives the user a hint or a clue about a certain color, such as “What is the color of a lemon?” The user has three tries to guess the color. If the user guesses the color correctly, the program congratulates him and tells him how many guesses he used. If he doesn’t guess the color in three tries, the program tells him what the color was.

The program should be built using two vectors, one for the colors and one containing the hints to the colors. Declare your two vector objects and use the *push_back()* function six times to fill each vector (six for hints, six for colors) There should be a “play loop” allowing the three guesses. Use the *rand()* function to generate a number between 0 and 5. Use this value as the selected color/hint. (You’ll use the number to reference the color and hint vector values.) Remember, if the user does not guess the color in three tries, the program tells the user what the color was. Be sure your user interface gives your user clear instructions for your game.

44. This program gives you a lot of practice writing *for* loops. Write a C++ program that asks your user to enter the height and width of a rectangle, and a character symbol. Also ask if the user wishes to see a “hollow” rectangle or a “filled” rectangle. Use the symbol to draw the specified rectangle. NOTE: you will draw only one rectangle when you ask the user for rectangle information. You’ll get the info, draw the rectangle and ask if they wish to do another.

Use nested *for* loops to draw the desired rectangle. The outer *for* loop controls the output for each row and the inner *for* loop controls the output for each column. The inside region of the hollow rectangle should be drawn with blank characters. Here are two examples—one filled and one hollow:

Filled Rectangle = 4 rowsby 6 columns

```
&&&&&  
&&&&&&  
&&&&&&  
&&&&&&
```

Hollow Rectangle = 5 rows by 10 columns

```
&&&&&&&&&  
& &  
& &  
& &  
&&&&&&&&&
```

The rectangular height and width must be at least three rows tall, and five columns wide. Determine an upper limit for the width and height based on the limitations of your console window. You should choose a number that produces a rectangle that fits on the standard console screen. The title and “would you like to do another?” question should also be seen on the screen. Do not draw a rectangle that has invalid number of rows or columns.

Check that the user has entered valid data for the rectangle. If the user enters an invalid number for either the width or height, give an error message and drop to the code that asks to “do again.”

45. This program goes where no program has gone before, exploring new frontiers in C++. The program performs calculations concerning weight on various planets as well as travel time between planets. Your program should write out a program title and brief introduction. Ask the user to enter her name. Politely ask her how much she weighs (or would like to weigh) in pounds, the speed at which she wishes to travel (in miles per hour), and the planet that she wishes to visit. (You will need to provide a menu of some sort with the planet name and a way for the user to select her chosen destination.) Using this data and the information from the Table 3-8, calculate the user's weight on the planet and travel time from Earth. These equations might be useful:

$$\text{Weight On New Planet} = \text{Weight On Earth} * \text{Surface Gravity Of New Planet}$$

$$\text{Travel Time (hours)} = \frac{\text{Travel Dist (miles)}}{\text{Rate (mph)}}$$

Here is a weight calculation example for the planet Saturn:

$$\text{Weight On Saturn} = \text{Weight On Earth} * 1.17$$

In the program output write the traveler's name weight on Earth, destination planet, and weight on this planet. Also report travel time results showing total hours. Break down the travel time into years, days, and hours. You may use 24 hours in a day and 365 days in a year (ignore leap years).

Use vectors to hold planet names, distance from the sun, and the specific gravity values. The menu for selecting the planet should provide an easy way to reference the selected planet's data.

Your program should have only one set of result-variables. The calculations and resultant output should be written in only one place in your program. Do not duplicate the calculation and output code for each planet. Make your program as efficient as possible without losing clarity.

TABLE 3-8

Planet Information

Planet	Distance from Sun (millions of miles)	Surface Gravity as a Function of Earth's Gravity
Mercury	36	0.27
Venus	67	0.86
Earth	93	1.00
Mars	141	0.37
Jupiter	483	2.64
Saturn	886	1.17
Uranus	1,782	0.92
Neptune	2,793	1.44

- 46.** This C++ program performs a vote counting and tallying procedure. It presents the user with a question and a “ballot” of possible answers. It allows her to vote multiple times. After voting is complete, the program writes the results to the screen. For this program, write your name, the program title, and brief instructions to the screen. The program should ask the user to select a favorite item from a group of items (such as favorite dessert). Print a ballot of the choices available and have her select her favorite item. There should be five choices. Incorporate a loop so that she is allowed to vote multiple times. Keep counters for valid and invalid votes as well as for individual item votes. Note that the sixth menu item is the option to quit voting. This option eliminates the need to ask the user if she wishes to vote again. If she selects an invalid choice, print a message informing her that the choice is invalid. Once she has finished voting, show the results to the screen.

For example, if the opinion poll dealt with favorite desserts, the voting options might look like this:

```
***** FAVORITE DESSERT OPINION POLL *****
```

1. Key Lime Pie
2. Chocolate Cake
3. Cheesecake
4. Apple pie with ice cream.
5. I have a different favorite dessert.
6. Quit voting

Please choose your favorite dessert from the listing above by number.

If the user makes an invalid selection, that vote is not counted in the total votes. Once the voting has been completed, calculate the percentage received for each item. Print the total number of votes, individual votes for each item, and percentages in an easy to read table. You should also show the number of invalid votes received. Your data might look something like this:

```
*****FAVORITE DESSERT OPINION POLL RESULTS *****
```

ITEM	VOTES	%
Key Lime Pie	5	33.3
Chocolate Cake	3	20.0
Cheesecake	3	20.0
Apple pie with ice cream.	1	6.7
I have a different favorite dessert.	3	20.0
Total Valid Votes:	15	
Total Invalid Votes:	2	Total votes received: 17

- 47.** Write a complete C++ program entitled the Fruit Finder, that fills a vector with seven different types of fruit. Write your name and a program title to the screen. Have the program ask the user to enter a sentence regarding fruit. The program searches the sentence for any occurrence of one of the seven fruit types. If it finds one, substitute the “Brussels sprouts” for the fruit and write out the user’s new sentence. If it doesn’t find the fruit, report that the user must not like fruit. Be sure to test for the fruit being the first word in the sentence (i.e., capitalized). Here’s an example, suppose your fruit types were apples, bananas, peaches, cherries, pears, oranges, and strawberries. If your user entered:

“I really love peaches on my cereal.”

your program would find peaches and substitute Brussels sprouts. The new string is

"I really love Brussels sprouts on my cereal."

If the user entered

"I'd rather have a candy bar."

your program might write

"You must not enjoy fruit."

If the user entered

"Apples are wonderful with lunch."

your program would write

"Brussels sprouts are wonderful with lunch."

48. The C++ Pond Pump Manufacturers build pond skimmers that draws water into a filter box located at the surface of the pond. The skimmer action draws the surface water into the box and subsequently acts like a vacuum removing leaves from the pond's surface keeping the water crystal clear. It is important that the skimmer match the size of the pond. If it is too small, it is ineffective. If it is too large, it pulls the fish and plants into the filter. (This is very hard on the fish!)

This program helps the pond builder select the correct skimmer for her pond. The C++ Skimmers are based on the total surface area of the pond. The Class A Skimmer should be used with ponds that have at least two and no more than four square yards of surface area. The Class B Skimmer is for ponds up to eight square yards and the Class C Skimmer is for ponds up to fifteen square yards. (For larger ponds we suggest placing skimmers at various locations, so that the sum of the skimmer capacity meets the total square yardage. You can tackle this problem on your own.)

We'll have the user select the general shape of the pond and ask him for the dimensions. The user's pond shape choices should be rectangular or circular. Your program should calculate the square yardage and determine the correct C++ Skimmer. For now, if the user's pond is too large for the Class C Skimmer, or too small for the Class A, the program should just report this. Ask the user for the dimensions in inches (length, width, if rectangular, diameter if circular). Report the total surface area in square yards and required skimmer. (Remember, there are 36 inches in a yard.)

49. In the early 1900s, tank wagons were developed to haul fuel and tar. (Search on the web for Tank Wagons for more information and photos.) Some tank wagons were built on vehicle chassis and others on railroad car frames. Today, there are railroad tank wagons, truck wagons, and farm tank wagons with gross vehicular weights up to 96,000 pounds! The C++ Tank Wagon Manufacturing Company builds high quality wagons on both truck and trailer frames. This program will help our company design tank wagons that best fits our customers' needs.

Your program should ask the user to enter the name of his business, the required total tank size in gallons, what he will be hauling (food items, non-food items, hazardous materials), the number of compartments and the desired tank

material (aluminum or steel). Your program then determines a configuration for the user, including whether the tank can fit on a truck or on a distribution trailer. (It may be possible to build the wagon on either a truck or trailer frame.) Here are the restrictions: trucks are limited to tanks no larger than 5000 gallons, trailers should be at least 2000 gallons but are limited to 9000 gallons. Any tank requiring more than three compartments must be on a trailer. Steel tanks are required for hazardous materials. Obtain the user's information and write the suggested tank configuration(s) to the screen. Note: it is possible that our user specifications match both truck and trailer configurations (show both) or are invalid criteria for the tank wagons we manufacture. If this is the case, issue a "Sorry" message and ask if he wishes to design another tank wagon.

50. The C++ Tropical Fish Store is here to help you select the perfect tank for your tropical fish enjoyment. The rule of thumb for setting up a nice, healthy fish tank is two gallons of water for every inch of fish in the tank. (You must remember to consider the adult size of the fish!) This means if you have three adult fish, and each fish is 3" long, you need to have at least a eighteen gallon tank.

Another rule involves what sort of fish will live in the tank. If all of the fish are peaceful, then a tank sized according to total adult fish length will be fine—however, if there are aggressive fish in the tank, then the tank should be sized 1.5 times that. (In the real-fish world, you must be careful in setting up your tank and selecting your fish! Search for Tropical fish on the web, to see pictures and types of fish.) For the sake of this program, we'll pretend our rules are fine!

In this program we ask the user to enter the information for the fish they expect to place in the new fish tank, then we'll suggest the correct size tank. Tanks come in 20, 40, 60, and 80 gallon sizes. Set up three vectors, one for the type of fish (i.e., tetra, loaches, etc.), one for the number of each type of fish and one that holds the adult size of a single fish. (This means that the same element in the three vectors correspond to one type, size, and number of fish.) Use a *while* loop to collect the fish data. If the user enters "done" for the fish type, stop collecting data. Next ask your user if there are any aggressive fish in the group. Now determine the best sized tank based on the above rules. (If the user requires a tank larger than the 80 gallon size, tell the user that we don't carry a tank for that number of fish.) Your program needs to write out a tank report including a table showing the fish information, total fish, total length of fish, minimum required gallons, and the appropriate C++ tank. Here is a sample output for a tank with five Clown Loaches and five Neon Tetras:

C++ Tropical Fish Store Tank Requirements		
Type of Fish	Number	Adult Size
Clown Loaches	4	3"
Neon Tetras	5	1"
Aggressive: No	Total Fish Length: 17"	Min Tank Size: 40 gallons

51. The C++ Distillery produces an excellent whiskey that is blended from a combination of mash and corn, aged in oak barrels for at least two years. The whiskey is filtered through charcoal and bottled in clear two liter triangular-shaped bottles.

We're going to write a program to help our distillery owner order the correct number of bottles for his product. The oak barrels are cylindrical in shape and range in size. Ask your user to enter the name for the whiskey that is going to be bottled, the size and number of barrels, and their diameter and height (in inches). We'll assume the barrels are all the same size. Approximately five percent of the volume is lost during the aging and filtering process. Once you have the barrel information, calculate the amount of whiskey to be bottled and the number of bottles required for this batch of whiskey. Bottles come to the distillery in cases of twelve, so also report the number of cases our owner should order. Remember, 231 cubic inches is a gallon and a gallon is equivalent to 3.785 liters. Note: You need to order enough bottles so that all the whiskey can be bottled. You may have a few extra bottles. Also, any whiskey that is not bottled can be shared amongst the employees after the bottling is finished. Report that quality too.

52. Fuel efficiency is on everyone's mind these days. We're going to write a program to help the car buyer analyze the expected price of driving her new car. This program presents the user with a list of five types of cars and their expected miles per gallon (mpg). We'll ask her to select two of the vehicles and enter the number of miles driven each year. Also enter the average price for a gallon of gas. The program then reports the yearly total number of gallons required by each vehicle, total cost, and shows the difference in cost between the two vehicles. The program should use two vectors, one for the type of vehicle and the other for integers that hold the corresponding mpg values.



Functions

Part I: The Basics

KEY TERMS AND CONCEPTS

arguments
automatic variable
C++ libraries
call-by-reference
call-by-value
called function
calling function
call statement
flags
function
function body
function call
function header line
function declaration or prototype
global variable
input arguments
local variable
return statement
return type
static variable
variable scope

KEYWORDS AND OPERATORS

return
void

CHAPTER OBJECTIVES

Present the concept of a function—which is a modular block of code—in the C++ language.

Demonstrate the basic format for any C++ function.

Describe how to write functions and how to pass data between them.

Introduce the notion of variable scope—how long a variable is “alive” in a program.

Illustrate how the location of the variable declaration determines the scope of the variable.

Present many simple program examples to illustrate these basic function concepts.



Little Picture, Big Picture

In the beginning chapters of this text, we saw functions used three different ways. First, when we wrote our programs in Chapter 2 and 3, we used the `main` function to contain our program code. Next, we included various libraries or the standard namespace so that we could use their functions, such as the `rand()` and `rand()` when working with random numbers or `sqrt()` when we needed the square root of a value. Last, we called functions that belonged to classes—for example, when we added an element to a vector object, we used the `push_back()` function.

In this chapter, we learn the basic mechanics of writing functions. Our functions here will be standalone blocks of code that are designed to do a specific task (or tasks). For example, the `rand()` and `rand()` functions within the `cstdlib` library are standalone functions. In the “old days” of C programming, all C-based software was constructed with these types of functions. But in the programming world today, the world revolves around classes and objects. We have used the string and vector classes by creating our own objects and calling the class functions. We have also used the pre-defined objects in `iostream` (`cout` and `cin`). Don’t lose sight of our ultimate goal—to become experts at writing our own classes—and classes contain functions that do the tasks defined in the class “job description.”

Before we can begin writing our own classes, we need to write standalone functions. It is not a difficult task, learning to write functions, but you must pay attention to how the functions are set up. Then the jump from writing these types of functions to class functions is small. Now is the time to concentrate on the parts and the pieces, the prototypes and the calls, the input lists and the return values. Spending time here will pay off when we get to classes, I promise.

4.1

Functions in C++

A **function** is a complete section of C++ code with a definite start point, an end point, and its own set of variables. Functions can be passed data values and they can return data. Whether the function is standalone or part of a class, it should be designed so that it has one primary task to accomplish.

function

a discrete module or unit of code that performs specific tasks

Life with Just a *main* Function

At this stage in your study of C++ programming, you are familiar with data variables and arithmetic operations, if and *switch* statements, as well as *for*, *do while*, and *while* loops. We have also seen how to use the *cin*, *cout* and *getline* objects from *iostream*, and have used the string and vector classes. But our entire program has been contained with the *main* function. Program 4-1 shows a simple program that asks the user her name and age, and writes the information to the screen.

Program 4-1

```
1 //How old are you?
2 //Entire program contained in the main function.
3 #include <iostream>
4 #include <string>
5
6 using namespace std;
7
8 int main()
9 {
10    int age;
11    string name;
12
13    //Write a greeting
14    cout << "\n Hello from a C++ program! \n";
15
16    //Ask for the user's name
17    cout << "\n What is your name?  ";
18    getline(cin,name);
19
20    //Ask for age
21    cout << "\n How old are you?  ";
22    cin >> age;
23
24    //Write information
25    cout << "\n Hi " << name << "! You are "
26              << age << " years old.  \n";
27
28    return 0;
29 }
```

Output

```
Hello from a C++ program!
What is your name? Mary Jones
How old are you? 39
Hi Mary Jones! You are 39 years old.
```

Life with Functions

Nearly all of our C++ programs are more complicated than asking for a person's name and age. As a first look at how to write functions in C++, we'll re-write this program using functions. There are four separate tasks in Program 4-1, writing a greeting, asking for the user's name, age, followed by writing the information to the screen. In Program 4-2 these four tasks have been broken out into four separate blocks of code. These are functions. By breaking the different tasks into functions, we organize the program into regions that accomplish specific tasks. For now, examine Program 4-2 and its output. We'll dissect this program in the following section.

Program 4-2

```
1 //Simple Functions and How Old Are You?
2
3 #include <iostream>
4 #include <string>
5
6 using namespace std;
7
8 //Function prototypes (or declarations)
9 void WriteHello();
10 string AskForName();
11 int AskForAge();
12 void Write(string name, int age);
13
14 int main()
15 {
16     int age;
17     string name;
18
19     //Write a greeting
20     WriteHello();           //call, no inputs, no return value
21
22     //Ask for the user's name
23     name = AskForName();    //call, returns name
24
25     //Ask for age
26     age = AskForAge();      //call, value assign into age
27
28     //Write information
29     Write(name, age);       //call, pass name, age to Write
30
31     return 0;
32 }
33
34 //Function definitions follow main
35
36 //Write_Hello writes a greeting message to the screen
```



```
37 void WriteHello()
38 {
39     cout << "\n Hello from a C++ function!\n";
40 }
41
42 //AskForAge asks the user for age, returns age.
43 int AskForAge()
44 {
45     int age;
46     cout << "\n How old are you?  ";
47     cin >> age;
48     cin.ignore();    //remove enter key
49     return age;
50 }
51
52 //AskForName asks for the user's name
53 string AskForName()
54 {
55     string name;
56     cout << "\n What is your name?  ";
57     getline(cin, name);
58     return name;
59 }
60
61
62 //Write writes the name and age to the screen
63 void Write(string name, int age)
64 {
65     cout << "\n Hi " << name << "! You are "
66                 << age << " years old.  \n";
67 }
```

Output

```
Hello from a C++ function!
What is your name? Mary Jones
How old are you? 39
Hi Mary Jones! You are 39 years old.
```

Functions are Good

When writing a program, you should organize the code into logical building blocks. This may involve laying out classes whose job tasks are then described and defined with functions. (We'll learn how to write classes in Chapter 7.) A well-written function can be reused in many programs. Use descriptive names for functions and variables, write with an easy-to-read style, and trap for errors you might encounter. You will then be well on your way to writing clearly understood software.

C programmers concentrated on writing functions. The function is the basic programming unit in a C program. We are heading toward designing and writing classes and using objects. In object-oriented software, the basic programming unit for C++ is

the class. The class descriptions include data and functions. The class functions “work on” the data. Some might say that all roads lead to Rome—but in the C++ world, all roads lead to classes and objects. We need functions to understand classes—so let’s keep going!

But First, Three Important Questions

As you begin to design functions and assign tasks to them, it is important for you to ask yourself these three questions:

“Who’s job is it?” You should concentrate on making sure that each of your functions take care of business—that it handle all the tasks it is supposed to do. For example, if you write a function that reads all the program data from a file, that function should open the file, read the data, and close the file. It is that function’s job to take care of all the file business, including closing the file if pertinent. (Do not assume “someone else” will close the file.)

“What input values does the function need to do its job?” In other words, what do you have to give the function for it to take care of business? If your function is supposed to calculate the volume of a pond, you’d need to give it the pond dimensions. (It would be someone else’s job to determine the pond dimensions.)

“What will my function return to me?” That is, what sort of value will the function return to you when it has finished its task? If the function opened, read, and closed the data file, it may give you back a flag to indicate if everything went well, true—all is OK, or false—there was a problem. If your function had to calculate the volume of a pond, it would return a double or float value of the volume.

So let’s get going! The quicker we have a complete grasp of functions, the quicker we can jump into classes.

4.2

Functions: Basic Format

The basic format of a function in C++ is:

```
return_type function_name(input parameter list)
{
    // Body of function
}
```

All of our programs so far have had one function, the *main* function. The format has been:

```
int main()
{
    // Body of function
    return 0;
}
```

The **return_type** is the data type (such as int, float, or double) of the value returned from the function. (The *main* function returns an integer.) Functions in C++ can have only a single return data type. This data is returned to the program

return_type

the data type of the variable passed back to the calling function via a return statement

arguments

input values for a function; also known as *input argument* or *input parameters*

input parameters

input values for a function, as known as arguments

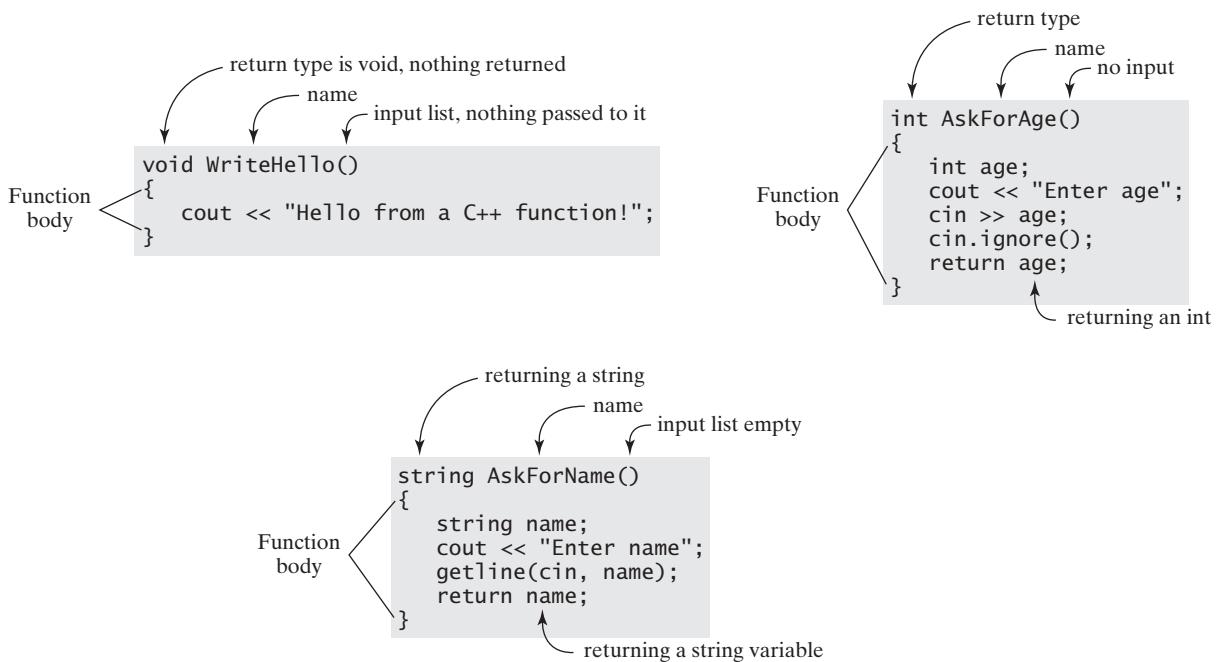
in a “return” statement. The *function_name* is the identifier (name) of the function and is used in the code to access the function. The programmer must follow the rules for naming variables when choosing a name for a function (see Chapter 2). The input data type and name list is the list of the input variable data types and names that the function receives. These inputs are referred to as **arguments** or **input parameters**. Figure 4-1 shows three of our NameAge program functions.

All functions in C++ follow the basic format presented above. The rules are simple. The function name must follow standard C++ naming conventions. There may be one return type. If there is no return type, the void data type is used. The input argument list must have data type and names (separated by commas), and you may pass in as many arguments as you like. If your list is empty (no arguments are passed), the parentheses are still required but will be empty () .

The following function, named *WriteHello()* is passed nothing and returns nothing:

```
void WriteHello()
{
    cout << "\n Hello from a C++ function!\n";
}
```

The *AskForAge()* function, shown below, does not need any input arguments, but it returns an integer value. Note that there is a return statement, “return age;” and the variable age is an integer. The word “return” is a C++ keyword. Also,

**Figure 4-1**

Basic C++ function format and the *Name* and *Age* functions.

because the `cin` statement leaves the Enter key in the input queue, we use the `cin.ignore()` function to remove that Enter key. This action falls under the realm of taking care of all business—who’s job is it to be sure that this Enter key is removed? It is `AskForAge()`’s job to make the call to `cin.ignore()` to insure there is not Enter key sitting in the input queue.

```
int AskForAge()
{
    int age;
    cout << "\n How old are you? ";
    cin >> age;
    cin.ignore(); //remove enter key
    return age;
}
```

The `AskForName()` function does not have any input arguments. It only asks the user to enter the name. The data is read into a string object using `getline` and the name is returned.

```
string AskForName()
{
    string name;
    cout << "\n What is your name? ";
    getline(cin, name);
    return name;
}
```

The last function’s job is to write the user’s name and age to the screen. We have to give it the name and age before it can do its job. Notice how the input parameter list has the two data items defined with their data types and variable names. Once the function has these two values (name and age), it uses `cout` to print the information to the screen. Since the function doesn’t give anything back, there is a void return type and no return statement. See Figure 4-2.

```
void Write(string name, int age)
{
    cout << "\n Hi " << name << "! You are "
        << age << " years old. \n";
}
```

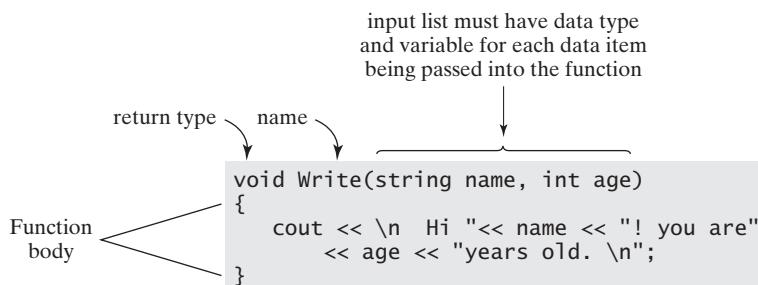


Figure 4-2

Input parameter lists must have a data type variable name list, separated by commas.

Calling and Called Functions

calling function

when one function uses (or invokes) a second function, the first function is the calling function

called function

when one function uses (or invokes) a second function, the second function is the called function

The terms **calling function** and **called function** are often used when referring to functions. When the one function (call it Function1) accesses another function (call it Function2), it is said that Function1 calls Function2. That is, Function1 is the calling function and Function2 is the called function. (Think of using the telephone to call your friend: you are the calling party; your friend is the called party.) It is quite common when a person is writing programs to have a function call a function, and it, in turn, calls another function.

In our Program 4-2, the *main* function calls the four other functions. In more complicated programs, especially with a Windows-based Graphic User Interface programs, there are long line of functions calling functions. The C++ Integrated Development Environments (IDEs) provide a Call Stack in the debugger so the programmer can trace the function calls. We don't need to worry about class stacks right now, if ever. Just be aware that this text refers to calling and called functions.

4.3 Requirements for Writing Functions

function prototype or function declaration

the C++ statement that contains the function name, input, and return data types

function definition or function body

the statements of C++ code that are contained inside a function, that perform the work of the function

function header line

the first line of a function, consisting of the return type, function name and function header line

call statement

the line of code in a program in which a function is invoked

There are sets of required statements whenever a programmer writes a C++ function. Every function must have a **function prototype** or **function declaration** statement, and a **function definition** or **function body**. The function prototype statement can be thought of as a model or a pattern that tells the compiler the function name, and its return and input types. The function definition contains the actual C++ code that performs the function's task. The first line of the function definition is the **function header line**. That line contains the return type, function name, and input parameter list. Whether your function is a standalone function or a function contained within a class, you must have a declaration (prototype) and definition for each function.

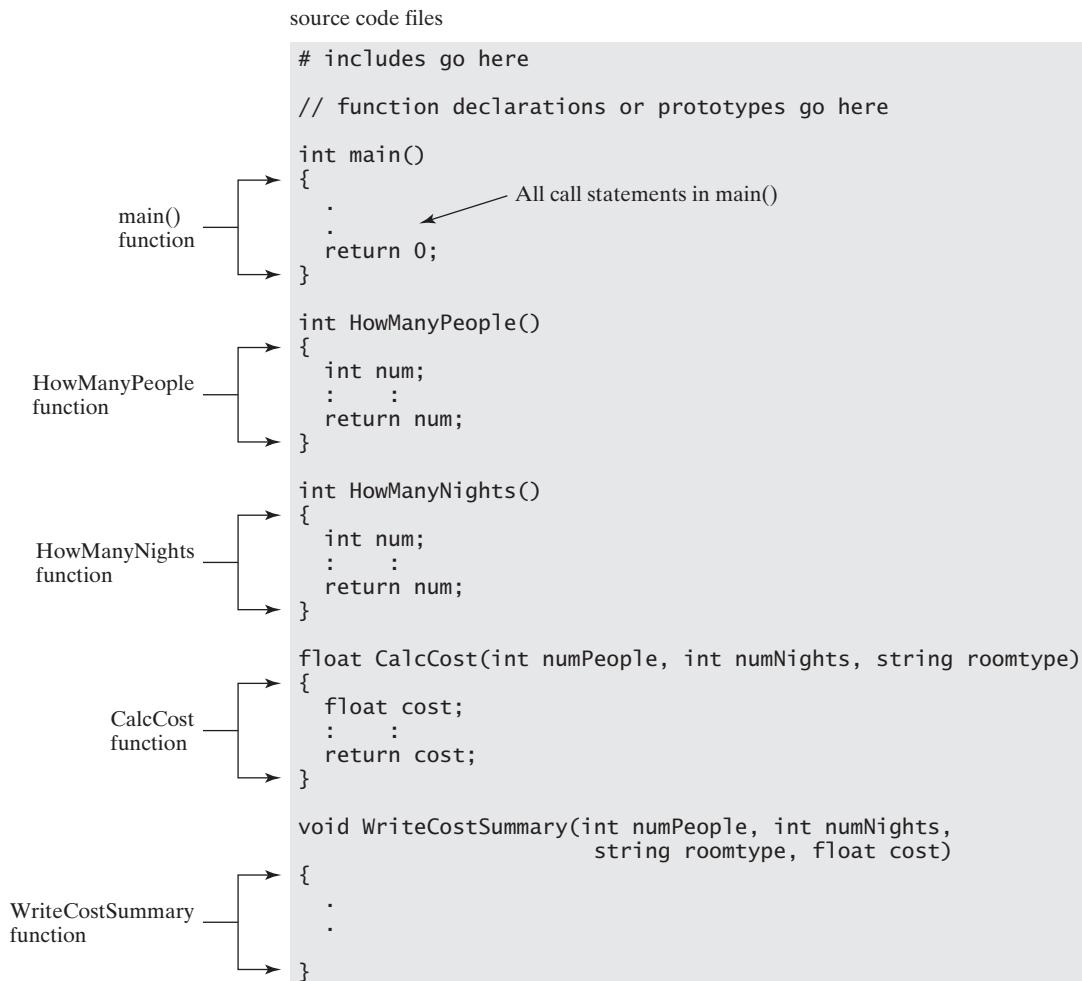
In order to invoke (make use of) the function, you'll need to have a statement that calls it. The **call statement** is the statement of C++ code located in the program where the function is to be used or accessed. Depending on the requirements for the function (input list) and return type—your function calls may vary in appearance. Don't worry, we'll examine the basics, trust that you'll get the hang of it quickly.

Would You Like to Stay at the C++ Hotel?

Need a vacation? A few days at the beach? Let's write another program that gathers information and calculates the cost of a room at the C++ Hotel. This first class hotel is located at a beautiful beach. There are three possible rooms available and the room rates are based on the view. The view of the beach costs \$99.95 per night, view of the swimming pool \$79.95, and the cheapest, but least desirable room is one with a view of the parking lot at \$69.95 per night. These base rates are for a single room with one person. Whatever room you choose, each additional person is \$10 per night.

In this program we set up the function declarations at the top of the file, as the compiler needs to see either the declarations or the actual functions before it sees the calls to them. We then write the *main* function and the function definitions are located after the *main* function. It is very important for you to realize the each function must be a discrete block of code with a function header line and body. See Figure 4-3 to see a framework of how these functions must be laid out in the source code file.

We first place the *#includes* at the top of the program and the function declarations above the *main* function header line. The *main* function calls the functions in the required order. The function definitions are located after the *main* function. Program 4-3 is seen on the next page.

**Figure 4-3**

Layout for the C++ Hotel program. Each function must be a discrete block of code.

Program 4-3

```
1 //A program that calculates the cost of a
2 //room at the C++ Hotel.
3
4 #include <iostream>           //for cout, getline, cin
5 #include <vector>
6 #include <string>
7 #include <iomanip>          //for setw()
8
9 using namespace std;
10
11 //Function declarations (prototypes)
12 void WriteGreeting();
13 int HowManyPeople();
14 int HowManyNights();
15 string WhatTypeRoom();
16 float CalcCost(int people, int nights, string roomType);
17 void WriteCostSummary(int people, int nights,
18                         string roomType, float cost);
19
20
21 int main()
22 {
23
24     int numPeople, numNights;
25     float totalCost;
26     string roomType;
27
28     //Calling the functions
29
30     //Ask for data
31     numPeople = HowManyPeople();
32     numNights = HowManyNights();
33     roomType = WhatTypeRoom();
34
35     //Calculate cost
36     totalCost = CalcCost(numPeople, numNights, roomType);
37
38     //Display results
39     WriteCostSummary(numPeople, numNights, roomType, totalCost);
40
41     return 0;
42 }
43
44
45 //obtain the number of guests
46 int HowManyPeople()
```



```
47  {
48      int num;
49      cout << "\n How many people will be staying with us? ";
50      cin >> num;
51      cin.ignore();           //remove the Enter key
52
53      return num;
54  }
55
56 //obtain the number of nights
57 int HowManyNights()
58 {
59     int num;
60     cout << "\n How many nights are you staying? ";
61     cin >> num;
62     cin.ignore();
63
64     return num;
65 }
66
67 //find out what type of room
68 string WhatTypeRoom()
69 {
70     vector<string> vRooms;
71
72     vRooms.push_back("parking lot view");
73     vRooms.push_back("swimming pool view");
74     vRooms.push_back("beach view");
75
76     cout << "\n What type of room? ";
77     for(int i = 0; i < vRooms.size(); ++i)
78         cout << "\n " << i+1 << setw(20) << vRooms.at(i);
79
80     cout << "\n Enter your choice here ==> ";
81
82     int whichOne;
83     cin >> whichOne;
84     cin.ignore();
85
86     return vRooms.at(whichOne-1);
87 }
88
89 //calc total room cost based on number of guests and room
90 float CalcCost(int people, int nights, string roomType)
91 {
92     float totalCost, rate;
93     float addPersonCost = static_cast<float>(0.0);
94 }
```

```
95     if(people > 1)
96         addPersonCost = static_cast<float>(10.00) * (people-1);
97
98         //room cost:
99         //parking lot view $69.95/night
100        //swimming pool view $79.95/night
101        //beach view $99.95/night
102
103        if(roomType == "parking lot view")
104            rate = static_cast<float>(69.95);
105        else if(roomType == "swimming pool view")
106            rate = static_cast<float>(79.95);
107        else if(roomType == "beach view")
108            rate = static_cast<float>(99.95);
109
110        totalCost = (rate + addPersonCost) * nights;
111
112        return totalCost;
113    }
114
115    //display a hotel program greeting
116    void WriteGreeting()
117    {
118        cout << "\n Welcome to the C++ Hotel Rate Program";
119
120    }
121
122    //write all info to the screen
123    void WriteCostSummary(int numPeople, int numNights,
124                                string roomType, float cost)
125    {
126        //Set precision to see $xxx.xx format
127        cout.setf(ios::fixed);
128        cout.precision(2);
129        cout << "\n C++ Hotel Rate Information: "
130                << "\n Your cost for a " << roomType << " room for "
131                << numPeople << " people for " << numNights
132                << " nights is $ " << cost << endl;
133
134 }
```

Output with 3 guests, 2 nights, in a beach-view room

Welcome to the C++ Hotel Rate Program

How many people will be staying with us? 3

How many nights are you staying? 2

What type of room?

1 parking lot view

2 swimming pool view

3 beach view

```
Enter your choice here ==> 3
C++ Hotel Rate Information:
Your cost for a beach view room for 3 people for 2 nights is $ 239.90
```

Concentrate on the Function First

When new C++ programmers get started writing functions, they often try to do everything at once! They will write the prototypes and then start on main, and as soon as they reach the call statement, they begin writing the function! OH! That's too much to keep track of for any programmer. On paper, it is an excellent idea to lay out the various functions and think about how you will write them. Also, think about main and the flow of the code. Once you are ready to sit down at the computer, write one prototype and its function. Then see if you can call it correctly and get the correct results!

When you tackle each function, just concentrate on that function. Work as if you have blinders on and can not see any other part of your program. Each function is its own block of code, with its own variables and control statements. Remember to ask yourself the three function questions: 1) what will this function do? 2) what value(s) does it need to do its job (i.e., inputs), and 3) what will the function return? Taking this approach with writing function will serve you well.

Function Declaration or Prototype

Except for the *main* function, all functions used in a C++ program must have a function prototype statement. (The *main* function does not require a prototype because the operating system automatically looks for the *main* function when the program is executed.) The prototype statement is a declaration statement that provides the compiler with information about the function name, input, and return data types. Variable names can be included in the input parameter list—we will write all of our prototypes in this manner. Before a function can be called, the calling function must know about the called function. The prototype may be declared in the calling function before the call, it can appear above the calling function (such as in the C++ Hotel program), or it can appear in an include file. The important point is that the compiler must have seen the function prototype before the function is called. “Imagine trying to find an apple in a grocery store if you don’t know what an apple is.” To clarify why it has to see the prototype 1st.

The form of the function prototype is:

```
return_type function_name(input parameter type list);
```

Here are the four prototypes in the C++ Hotel program. These statements are in the source code above the *main* function so that they are read by the compiler before it reaches *main*.

```
//Function declarations (prototypes)
//no inputs, no return value
void WriteGreeting();

//no inputs, returns an integer value
int HowManyPeople();
```

```

int HowManyNights();

//no inputs, returns a string object
string WhatTypeRoom();
//three input values, 2 ints, 1 string, returns a float
float CalcCost(int people, int nights, string roomType);

//four input values, no return value
void WriteCostSummary(int people, int nights,
                      string roomType, float cost);

```

Function Definitions, Function Header Lines, and Function Bodies

Once the programmer has defined the name, the input list and return type for a function in the prototype, he can then concentrate on writing the actual function. The function header line is the first line of the function, and will be a duplicate of the prototype statement (assuming variable names have been listed with the input types). Your copy/paste feature will help you in this typing endeavor. Recall that the function header line is the first line of the function. Function header lines have the return type, function name, and input parameter list.

In the *CalcCost* function the input parameter list has three input values, two integers and one string. When you write the input list with the data type and variable name, this declares those variables that belong to that function. Your prototype and function header line should look the same—with the same variable names. Notice how inside the *CalcCost* function we have and use the variables “people,” “nights,” and “roomType.” (Don’t be concerned that we have “num” and “numPeople” and “numNights” other places in our program. Remember, as you write this function don’t worry about anything else in our program!)

```

float CalcCost(int people, int nights, string roomType)
{
    float totalCost, rate;
    float addPersonCost = static_cast<float>(0.0);

    if(people > 1)
        addPersonCost = static_cast<float>(10.00) * (people-1);

    if(roomType == "parking lot view")
        rate = static_cast<float>(69.95);
    else if(roomType == "swimming pool view")
        rate = static_cast<float>(79.95);
    else if(roomType == "beach view")
        rate = static_cast<float>(99.95);

    totalCost = (rate + addPersonCost) * nights;
    return totalCost;
}

```

Return Statement The **return statement** serves three purposes in a C++ function. First, it is required when the function is returning a value to the calling function. If the function is not returning a value, the return statement is not required. Second, many return statements may be included in a function. The return statement causes the program control to exit the function and to return to the calling function. It is possible to “bail out” of a function by using a return statement. (See the *HowMuchFood* function below.) Third, an expression may be evaluated within the parentheses format of the return statement. The return statement in *CalcCost* is:

```
return totalCost;
```

but it could be written like this:

```
return (rate + addPersonCost) * nights;
```

The function *WriteGreeting* function does not return a value to the calling function, so no return statement is required:

```
void WriteGreeting()
{
    cout << "\n Welcome to the C++ Hotel Rate Program";
}
```

The return statement also provides the programmer a way to terminate the function and return to the calling function at any point in the function body. The return statement can be used to return a value or simply to exit the function if nothing is returned. Leaving our C++ Hotel program for a moment, the function *HowMuchFood()* shows an input list with an integer representing the type of animal and a floating point value for the animal’s weight. These values are passed into the function. The required amount of food per day, in pounds, is returned. The return statements are located within the switch statement.

```
float HowMuchFood(int animalType, float weight)
{
    switch(animalType)
    {
        case 1: //elephant
            return(weight * 0.2); //no break needed- -exiting function
        case 2: // dog
            return(weight * 0.25);
        case 3: //squirrel
            return(weight * 0.4);
    }
}
```

If a single return statement is located at the end of the *HowMuchFood* function, break statements are required.

```
float HowMuchFood(int animalType, float weight)
{
    float total;
```

return statement

a line of code where program control leaves a function; an exit point from a function

```

switch(animalType)
{
    case 1: //elephant
        total = weight * 0.2;
        break;
    case 2: // dog
        total = weight * 0.25;
        break;
    case 3: //squirrel
        total = weight * 0.4;
        break;
}
return total;
}

```

Function Calls

Now that we have our functions written, we can make use of them via a call statement. When we examine the *main* function in this program, we see that it is very tidy and well organized. The ***call statement*** is the C++ statement (located in the calling function) where the called function is accessed. When a function is called, control is passed to the called function, and the statements inside that function are performed. (Copies of the data in the calling function are passed to the called function. More on this in a moment.) Control is returned to the calling function when the function tasks are completed. The call statement requires that just the variable names be used.

The six function calls in the C++ Hotel program are:

```

//WriteGreeting
WriteGreeting();
//Ask for data
numPeople = HowManyPeople();
numNights = HowManyNights();
roomType = WhatTypeRoom();

//Calculate cost
totalCost = CalcCost(numPeople, numNights, roomType);

//Display results
WriteCostSummary(numPeople, numNights, roomType, totalCost);

```

No Inputs and No Return Value When a function does not have any inputs, nor does it return anything to the calling function, the call statement is very simple. The *WriteGreeting* function is like this, and must be called like so:

```
WriteGreeting();
```

No Inputs but has a Return Value When a function does not have any inputs, but it does return a value, you have to be sure the call statement has an assign operator so that the returned value is placed in a variable. In this program, we have three functions of this type. Note how the input list parentheses are empty, but the return values are each assigned to one of main's variables.

```
numPeople = HowManyPeople();
numNights = HowManyNights();
roomType = WhatTypeRoom();
```

Input and No Return Values When a calling function must pass information to the called function, but the function doesn't return anything, the call statement does not have an assign operator. When this program writes the resultant cost summary to the screen, this task is done by the *WriteCostSummary* function. We pass it all the info, and it writes the data using cout. The function doesn't return any value to us.

```
WriteCostSummary(numPeople, numNights, roomType, totalCost);
```

Input and Return Values When a calling function must pass information to the called function, and that function returns a value to us, we need to have an assign statement to obtain that value. In the function *CalcCost*, we are passing a copy of the variable *numPeople*, *numNights*, and *roomType* to the function. This function is returning the cost value to us. Here is the call to *CalcCost*, which returns to us the total cost of the stay.

```
totalCost = CalcCost(numPeople, numNights, roomType);
```

Call-by-Value

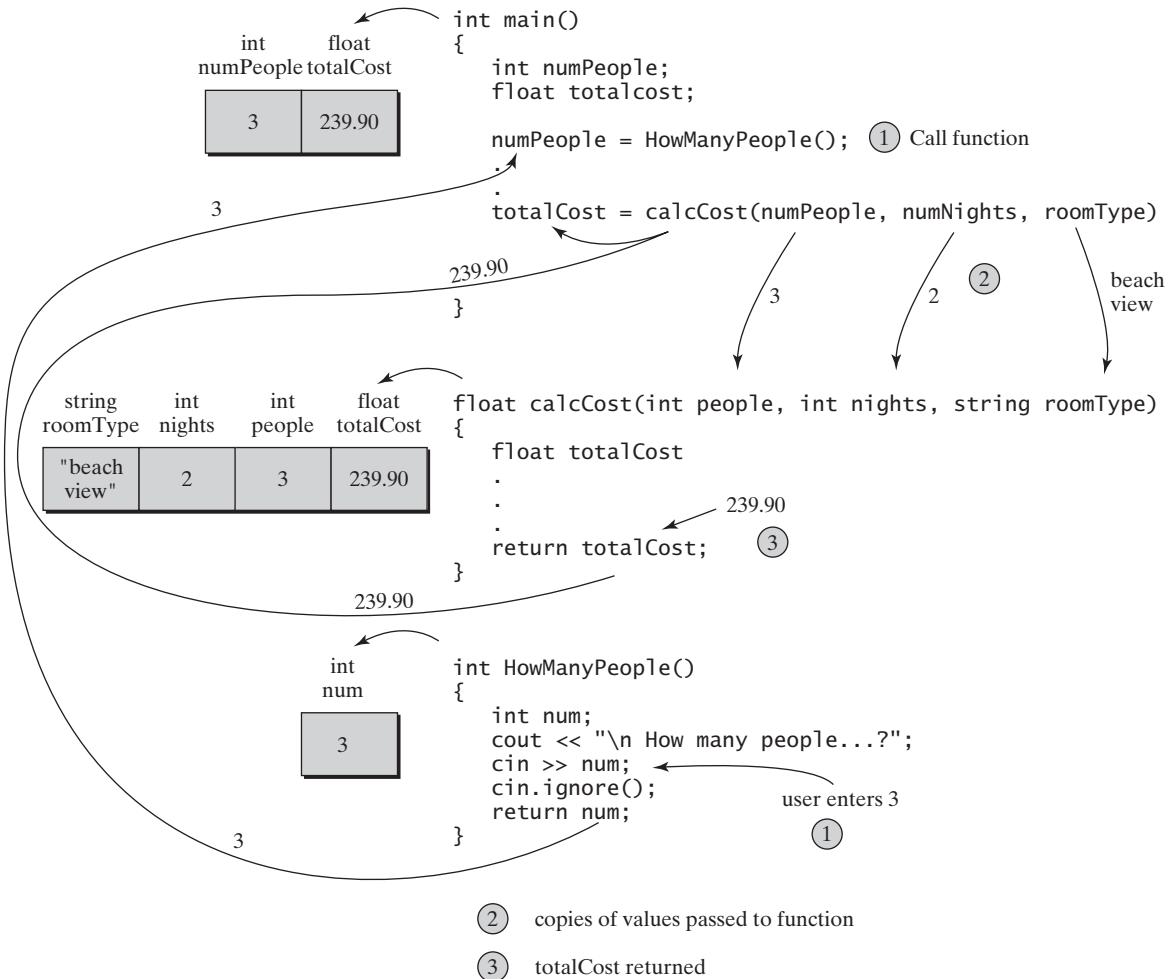
In these function examples, the data values are passed to and from the functions, and the functions actually have their own copy of the data. When the data is passed to a function, the value of the variable is copied into the input parameter variables of the called function. This type of function call is known as a **call-by-value**. When data is passed using a variable name in the call statement, the value of that data is copied into the variables of the called function. In the C++ Hotel program, each function has its own copy of the variables that it needs. This concept is illustrated in Figure 4-4. Notice how the *main* function calls its number of guests “*numPeople*,” *HowManyPeople()* has a “*num*” variable, and the *CalcCost()* has a “*people*” variable. As the program executes, all three of these variables contain the value for the number of guests staying in the room—but because each function is its own entity, the three variables are separate too.

Program 4-4 illustrates this function variable “ownership” concept again. In this program we ask the user to enter a number and use it as a limit for summing values from 1 to that value. That is, if the user entered 5, the program adds $1 + 2 + 3 + 4 + 5$, for a total of 15. The Add Values from 1 to N program has a *main* function that calls two functions, *AskForNumber* and *Add_1_to_N*. The user's number is passed to the *Add_1_to_N* function, which adds the numbers consecutively from 1 to the user's value. The *main* function then writes the results to the screen.

In this program, note that the variable names are not the same from function to function. The *main* function has the variables “*x*” and “*sum*.” But in the *Get_Number* function, the variable holding the user's number is “*number*” and in the *Add_1_to_N*, it is just “*n*.” The *sum* value in *main* is represented as the total value in the *Add_1_to_N* function.

call-by-value

term used when the value of a variable is passed to a function

**Figure 4-4**

Each function has its own copy of its variables. In the C++ Hotel program, the number of people and cost are illustrated here.

When variables are declared in a function header line or within the function itself, the variables are local to (are “owned by”) that function. They are not seen by any other function. Figure 4-5 illustrates this concept.

Program 4-4

```

1 //A program that asks the user to enter a value (N),
2 //and then we sum the numbers from 1 to N
3
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;

```

Each function has its own copy of the data variables.

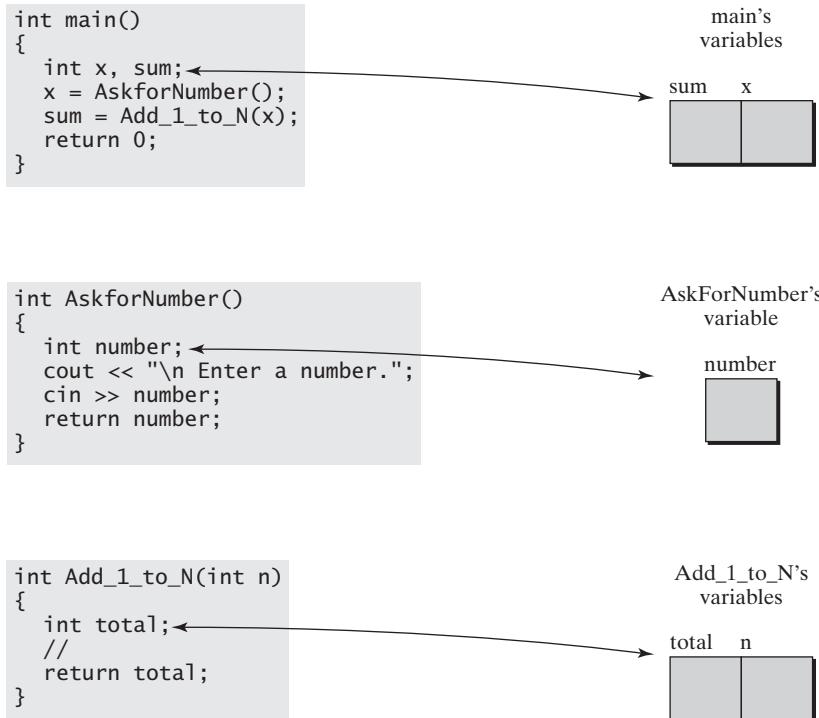


Figure 4-5

Functions have their own copies of the variables. These variables are referred to as local variables.

```

7
8
9 //Function Prototypes
10 int Get_Number();
11 int Add_1_to_N(int);
12
13 int main()
14 {
15     int x,sum;
16
17     x = Get_Number();      //call to function to get the user's number
18     sum = Add_1_to_N(x);   //pass x, adder returns the sum
19
20     cout << "\n The result from adding 1 + 2 + ... + " << x <<
21         " is " << sum << endl;
22
23     return 0;
24 }
25

```

```

26 int Get_Number()           //Function header line
27 {
28     int number;
29     cout <<"\n Enter a number ";
30     cin >> number;
31     return number;
32 }
33
34 int Add_1_to_N(int n)
35 {
36     int total = 0,i;
37
38     for(i = 1; i <= n; ++i)
39     {
40         total = total + i;
41     }
42     return total;
43 }
```

Output with input value of 42

Enter a number 42

The result from adding 1 + 2 + . . . + 42 is 903.

Recommendation! New C++ programmers, when presented programs that have different variable names in different functions for the same program value always ask, “Why would you code like this? Why not use the same variable name for the same program item?” The answer is “Of course! Use the same variable name!” But it is very important that students understand that each function has its own separate variables, and the variables can have different names.

Troubleshooting: Undeclared Identifier

A common mistake that many C++ programmers make is to think that once a data variable has been declared inside the *main* function, it can be seen by all the other functions. Remember, that each standalone function must have its own data variables. Look at Program 4-5, a re-written version of the Name and Age program from Program 4-2. Then examine the compiler errors and see if you can determine the problem.

Program 4-5

```

1 //Simple Functions and How Old Are You?
2 //This does not compile!
3
4 #include <iostream>
5 #include <string>
6
7 using namespace std;
```



```
8 //Function prototypes (or declarations)
9 void WriteHello();
10 string AskForName();
11 int AskForAge();
12 void Write(string name, int age);
13
14 int main()
15 {
16     int age;           //declare our variables here
17     string name;
18
19     //Write a greeting
20     WriteHello();
21
22     //Ask for the user's name
23     name = AskForName();
24
25     //Ask for age
26     age = AskForAge();
27
28     //Write information
29     Write(name, age);
30
31     return 0;
32 }
33
34 //Function definitions follow main
35
36
37 //Write_Hello writes a greeting message to the screen
38 void WriteHello()
39 {
40     cout << "\n Hello from a C++ function!\n";
41 }
42
43 //AskForAge asks the user for age, returns age.
44 int AskForAge()
45 {
46     cout << "\n How old are you? ";
47     cin >> age;
48     cin.ignore();    //remove enter key
49     return age;
50 }
51
52 //AskForName asks the user's name
53 string AskForName()
54 {
55     cout << "\n What is your name? ";
```



```

56     getline(cin, name);
57     return name;
58 }
59
60
61 //Write writes the name and age to the screen
62 void Write(string name, int age)
63 {
64     cout << "\n Hi " << name << "! You are "
65             << age << " years old. \n";
66 }
67

```

Compiler errors

```

asknameagefunc.cpp(47) : error C2065: 'age' : undeclared identifier
asknameagefunc.cpp(56) : error C2065: 'name' : undeclared identifier

```

4.4 Overloaded Functions

overloaded functions

functions that have the same name but different input parameter lists

C++ allows the programmer to create **overloaded functions**, meaning that it is possible to have two or more functions with the same name but with different input parameter lists. C++ does not allow overloaded functions to differ only in their return type. Ideally, your overloaded functions all have perform essentially the same job—but there are different input possibilities. One more thing, sometimes when you write overloaded functions, your program won’t use all of them. We may overload functions to give us several possible functions, and due to some condition, we don’t use all of them. This is especially true when we get to classes.

You are probably wondering what do we mean, overloading our functions? Let’s look at an example, and you’ll see how easy it is. In Program 4-6 we have four functions that are all named *SayGoodnight*. The task for these functions is to say goodnight, but in slightly different ways. Notice how we have four different prototypes and four separate functions. When you overload functions, you write each function normally—that is, you write the prototype and the function definition. For this program we call all four in main just so we can illustrate overloaded functions. (Remember when we write text like this: “Bob”, C++ interprets that as a string.) Look at the code in the *SayGoodnight* program to see that we call all four of these overloaded functions. Note: often programmers write several overloaded functions for completeness, but do not expect that you’ll be calling each of the overloaded functions in a program.

Program 4-6

```

1  //Using overloaded functions to say goodnight.
2
3 #include <iostream>
4 #include <string>
5

```

```
6  using namespace std;
7
8 //four different prototypes, same function name
9 void SayGoodnight();
10 void SayGoodnight(string name1);
11 void SayGoodnight(string name1, string name2);
12 void SayGoodnight(int number);
13
14 int main()
15 {
16     //call all four overloaded functions
17     SayGoodnight();
18     SayGoodnight("Bob");
19     SayGoodnight("Susan", "Melissa");
20     SayGoodnight(5);
21
22     cout << "\n All done saying Goodnight! \n";
23     return 0;
24 }
25
26 //a standard goodnight message
27 void SayGoodnight()
28 {
29     cout << "\n Goodnight! Sleep tight. "
30             << " Don't let the bed bugs bite. ";
31 }
32
33 //say goodnight to one person
34 void SayGoodnight(string name1)
35 {
36     cout << "\n Goodnight, " << name1 << "!";
37 }
38
39 //say goodnight to two people
40 void SayGoodnight(string name1, string name2)
41 {
42     cout << "\n Goodnight, " << name1 << " and "
43             << name2 << "!" << endl;
44 }
45
46 //say goodnight a number of times
47 void SayGoodnight(int number)
48 {
49     for(int i = 0; i < number; ++ i)
50         cout << "\n Goodnight!";
51 }
```



Output

```
Goodnight! Sleep tight. Don't let the bed bugs bite.  

Goodnight, Bob!  

Goodnight, Susan and Melissa!  

Goodnight!  

Goodnight!  

Goodnight!  

Goodnight!  

Goodnight!  

Goodnight!  

All done saying Goodnight!
```

4.5**Default Input Parameter List Functions****default input parameter list**

default values supplied in the function prototype; used if no values are passed to the function

The C++ language provides the programmer a shortcut way to assign values to the function inputs with the use of the **default input parameter list**. This useful C++ feature allows the program to declare a function that has a variable number of input parameters, and the programmer also supplies the default values for the inputs. If the function is called and the values are not passed to it, the function simply uses the default values supplied in the declaration. The prototype for this type of function may or may not contain the variable name, but the default value must be supplied. In all of our examples, we supply the variable name in the prototype.

In the DrawLines program, shown as Program 4-7, a default parameter list is declared for a function that prints lines of characters to the screen. The function's job is to draw a given number of lines that are a certain length, with a given symbol. That is, when the function is called, it needs to know how many lines, what is the symbol, and how many symbols on each line. For this program, we set up the default values for the function to be a percent sign '%' for the symbol, twenty-five characters on a line, and the default number of lines is one. The function prototype is:

```
void DrawLines(char symbol = '%', int numOfSymbols = 25,  
               int numOfLines = 1);
```

Default parameter list functions require the programmer to pass inputs in the declaration order. Starting with the rightmost value, parameters may be omitted from the call. In the DrawLines program, we call the function in the following four ways.

```
DrawLines();           //no inputs, default values are used %, 25, 1  

DrawLines('@', 30);  //use @, and 30 chars, default value of 1  

DrawLines('#',15,3); //use #, 15 chars on 3 lines  

DrawLines('%',6);    //need to vary the second value, must have the first
```

The function must be called with input values starting on the left, and input values may not be skipped. If we want to use the default symbol and change the number of symbols, we need to call the *DrawLines* function like this:

```
DrawLines('%', 6);      // symbol must be passed with number of symbols
```

The function cannot be called in this manner:

```
DrawLines(, 6);           // Not a valid call :-(
```

If we wish to call the function using the default parameters for the symbol, and the number of symbols, but change the number of lines, the call must look like this:

```
DrawLines('%', 25, 4);   // symbol must be passed in with number of symbols
```

not like this:

```
DrawLines(, ,4);         // :-( invalid call
```

When working with default parameter list functions, you should place the value(s) you expect to change in the left side of the list and the values you seldom change on the right side. Since the values in the call may be omitted, beginning from the right, you can design your function to shorten your call list.

Program 4-7

```
1 //DrawLines function has a
2 //default parameter input list.
3
4 #include <iostream>
5 using namespace std;
6
7 //The prototype contains the default values.
8 void DrawLines(char symbol = '%', int num0fSymbols= 25,
9                 int num0fLines = 1 );
10
11
12 int main()
13 {
14     cout << "\n The DrawLines Program \n\n";
15     cout << "Default Line 25 % on 1 line";
16     DrawLines();
17     cout << "\n Change to 30 @ on 1 line";
18     DrawLines('@', 30);
19     cout << "\n Now draw 15 # on 3 lines";
20     DrawLines('#',15,3);
21     cout << "\n Last line has 6 % on 1 line";
22     DrawLines('%',6);
23
24     cout << "\n\n No more lines for you. \n";
25
26     return 0;
27
28 }
29
30 //The function definition does not contain the default values.
```

```

31 void DrawLines(char symbol, int numSymbols, int numOfLines )
32 {
33     int i, j;
34     cout << "\n";
35     for(i=0; i < numOfLines; ++i)
36     {
37         for(j=0; j< numSymbols; ++j)
38         {
39             cout << symbol;
40         }
41         cout << "\n";
42     }
43 }
```

Output

The DrawLines Program

Default Line 25 \% on 1 line

%%%%%%%%%%%%%%

Change to 30 @ on 1 line

@@@@@@@@cccccc@cccccccccccccccccccc@

Now draw 15 \# on 3 lines

#####

#####

#####

Last line has 6 \% on 1 line

%%%%%

No more lines for you.

4.6 Local, Global, and Static Variables

When you write standalone functions (as we have done so far in this chapter) the C++ variables within a function are hidden from all other parts of the program and cannot be accessed by any other functions. The only way to get to code or variables in a function is by calling the function and having the program control enter through the function header line. It is impossible to perform a jump or “goto” into the middle of a function.

Now that we have started working with functions, it is time to introduce the concept of variable scope. **Variable scope** determines which variables in the program are visible to other portions of the program as well as dictating “how long” a variable is available while the program is executing. This variable visibility issue includes the ability for one portion of the code to access a variable in other parts of the code. The scope of the variable is determined by the location where the variable is declared.

variable scope

the length of time that a variable is in existence as the program runs; it determines who can see or access the variable

Local Variables

The variables declared within the function and in the function header line input list belong to, or are “owned by,” that function. No other function can see, access, or change these variables; they are known as ***local variables*** or ***automatic variables***. These local variables come into existence when the function begins executing and are destroyed once program control exits the function. In the Add Values from 1 to N program example, each function has its own variable that represents the user’s number, but each function has a different name for this number. Most programmers will use the same name for a variable throughout the program—this action leads to consistency and easily understood code.

Block Scope

The concept of ***block scope*** illustrates how variables exist within functions and within blocks of code within functions. Let’s explain this concept in simple terms. When you declare a variable inside a set of braces {} that variable then exists as long as the program is executing inside that block of code. Once code execution moves beyond those braces, the variable goes out of scope. Figure 4-6 illustrates a very short program that contains local and global variables. It also shows the regions where the variables are in scope.

Global Variables

It is possible to declare a variable to be global. A ***global variable*** is declared outside any function and is visible to all functions in that program file; that is, all the functions can use and change a global variable. (If the global variable is

local variables

variables declared either inside a function or in the function header line

automatic variable

variables that are declared inside functions; also known as *local variables*

block scope

variables declared within a set of {} are in scope as long as program execution is within the {}

global variable

variable that is declared outside any function and all functions within the file can access it

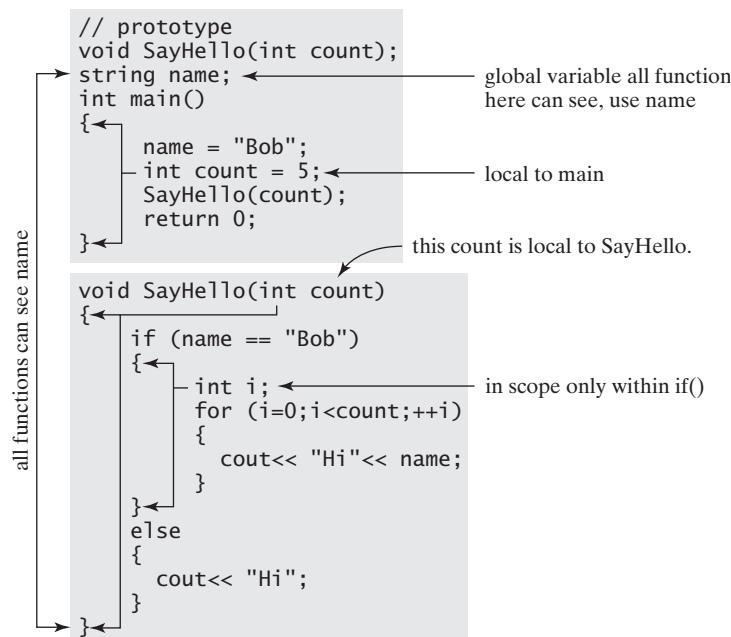


Figure 4-6

A program illustrating local and global variables. The `name` variable is global and all functions can see it.

needed in other files, it must be extern'd to that file.) Program 4-8 is a re-written version of our Name and Age program. In this program we declare the age and name variables above main. The four functions can access these variables directly, and there is no need to pass them between functions (as we did in Program 4-2).

Program 4-8

```
1 //Simple Functions and How Old Are You?
2 //Global variables
3
4 #include <iostream>
5 #include <string>
6
7 using namespace std;
8
9 //Functions do not pass or return anything
10 //since they are working with global variables.
11 void WriteHello();
12 void AskForName();
13 void AskForAge();
14 void Write();
15
16 //global variables
17 //all functions in this file can see them
18 int age;
19 string name;
20
21 int main()
22 {
23     //Write a greeting
24     WriteHello();
25
26     //Ask for the user's name
27     AskForName();
28
29     //Ask for age
30     AskForAge();
31
32     //Write information
33     Write();
34
35     return 0;
36 }
37
38
39 //Write_Hello writes a greeting message to the screen
40 void WriteHello()
41 {
```



```
42     cout << "\n Hello from a C++ function!\n";
43 }
44
45 //AskForAge asks the user for age, returns age.
46 void AskForAge()
47 {
48     cout << "\n How old are you?  ";
49     cin >> age;
50     cin.ignore();    //remove enter key
51 }
52
53 //AskForName asks the user's name
54 void AskForName()
55 {
56     cout << "\n What is your name?  ";
57     getline(cin,name);
58 }
59
60 //Write writes the name and age to the screen
61 void Write()
62 {
63     cout << "\n Hi " << name << "! You are "
64             << age << " years old.  \n";
65 }
```

Output

```
Hello from a C++ function!
What is your name? John Hancock Thomas
How old are you? 45
Hi John Hancock Thomas! You are 45 years old.
```

Global Variables are Hazardous

When you examine the code in Program 4-8 you may think, “HEY this is a great idea! It makes it so easy to use variables and functions.” You may think that life is better if there is no need to pass variables between functions. There are times and places for using global variables, but now is not the time, nor the place. Using global variables can lead to trouble, especially when programs are large or several programmers are working on one project. Programmers can run into naming conflicts, as well as crazy behavior because global variables are accessible by all functions, every function can change the value.

Global variables should be used sparingly and with considerable thought as well as subsequent documentation. As a general rule, global variables can be used for mathematical constants or universally used values and perhaps file pointers or stream objects. In some instances, global variables are necessary if functions are provided for the programmer, but the input parameter list is fixed. In this case, the variables must be global for the function to be able to access required values. If the program is quite large, with many user-written libraries, it is best to avoid global declarations altogether.

Troubleshooting: Global Variable *y0* and *y1* and *cmath*

The TakeASquareRoot program demonstrates how global variables can conflict with C++ libraries. This program uses the global variables “y0” and “y1.” The program needs the C++ cmath include file because it uses the square root function. C++’s cmath library includes C’s math.h file. As it turns out, the code in the math.h file has defined y0 and y1 variables. When this program attempts to declare global y0 and y1, the compiler thinks you are redefining y0 and y1.

Program 4-9

```
1 //This program does not compile.  
2 //There is a conflict using a global variable  
3 //and cmath library.  
4  
5 #include <iostream>           //for cout  
6 #include <cmath>              //for sqrt  
7 using namespace std;  
8  
9 //declare two global variables  
10 double y0, y1;  
11  
12 int main()  
13 {  
14     //assign value into y0  
15     y0 = 5000.0;  
16  
17     //take the square root of y0  
18     y1 = sqrt(y0);  
19  
20     //write result  
21     cout << "\n Square root of " << y0 << " is " << y1;  
22  
23     return 0;  
24 }
```

Compiler errors

```
HazardDriver.cpp(10) : error C2373: 'y0' : redefinition; different type  
modifiers  
    c:\ program files\ microsoft visual studio\ vc98\ include\  
        math.h(434) : see declaration of 'y0'  
HazardDriver.cpp(10) : error C2373: 'y1' : redefinition; different type  
modifiers  
    c:\ program files\ microsoft visual studio\ vc98\ include\  
        math.h(435) : see declaration of 'y1'.
```

The problem with this small program is that the compiler has already seen a `y0` and `y1` variable in the `math.h` file. The compiler believes we are trying to redefine `y1` in our program. It is possible to examine the contents of `math.h`. A few lines are:

```
_CRTIMP double __cdecl _y0(double);
_CRTIMP double __cdecl _y1(double);
_CRTIMP double __cdecl _yn(int, double);
```

By placing a global variable in our program, we have a conflict with variables in the math standard library. What is the moral of this story? Always use global variables sparingly, and carefully, lest you bring on more problems than you can imagine!

Static Variables

When a program begins to execute, the local variables inside the `main` function are created (that is, memory is allocated for them). The storage space for variables within other functions is not allocated until the individual function is called and the program control is passed to it. Once the function statements are completed and we exit, the variables that were declared inside the function are released from memory and the contents of these variables are lost.

Functions can retain the variable values by using the `static` specifier in the variable declaration. Once a variable is declared a **static variable**, the variable contents are retained until the program is terminated. The variable is still visible only to the function and it is not seen by any other parts of the program. The first time the function is entered, the static variable initialization occurs. If there is no initial value assigned in the code, C++ initializes the static variable to zero. The static variable initialization statement is performed only once while the program is running. The Shopping Program, Program 4-10 illustrates how static variables are used. If a function must “remember” a value, even after the program control has left that function, a static variable will keep its value until the program ceases to run.

static variable

local variable that retains its value until the program is terminated

In the Shopping Program, the function `HowMuchHaveYouSpent` has a static total. Each time the function is called, it asks the user the amount of the recent purchase. The amount is added to total, and total is returned to the calling function, `main`. Review the code and the output to see that total accumulates a running total of purchases. (What do you think the output would be if we didn’t declare `total` as `static`?)

Program 4-10

```
1 //This program asks the user to enter the
2 //amount of the item purchased.
3 //The function has a static variable, which
4 //maintains (remembers) the amount entered.
5
6 #include <iostream>
7 #include <string>
```

```
8
9  using namespace std;
10
11 //function declaration
12 float HowMuchHaveYouSpent();
13
14 int main()
15 {
16     float total;
17     string answer = "yes";
18
19     cout.precision(2);
20     cout.setf(ios::fixed);
21
22     while(answer == "yes")
23     {
24         total = HowMuchHaveYouSpent();
25
26         cout << "\n You've spent $ " << total << " so far today. ";
27
28         cout << "\n Keep shopping? yes/no ";
29         getline(cin,answer);
30     }
31
32     cout << "\n Your shopping trip today cost you $ " << total << endl;
33     return 0;
34 }
35
36 float HowMuchHaveYouSpent()
37 {
38     //total is set to zero 1st time in here
39     static float total = static_cast<float>(0.0);
40
41     float amount;
42     cout << "\n How much was this recent purchase? $ ";
43     cin >> amount;
44     cin.ignore();
45
46     //keep a running totof of the purchases
47     total = total + amount;
48     return total;
49 }
```

Output

How much was this recent purchase? \$ 5.56

You've spent \$ 5.56 so far today.

Keep shopping? yes/no yes

```
How much was this recent purchase? $ 39.21
You've spent $ 44.77 so far today.
Keep shopping? yes/no yes
```

```
How much was this recent purchase? $ 25.11
You've spent $ 69.88 so far today.
Keep shopping? yes/no no
```

Your shopping trip today cost you \$ 69.88

4.7

More Fun with C++ Classes, the stringstream Class

The C++ `stringstream` class is a useful class when working with string and numeric data. It is referred to as a stream object, and is found with the C++ input/output stream libraries. You need to include the `<sstream>` header when you want to use a `stringstream` object. In a nutshell, you can use the same formatting tools that we use with the `cout` object with a `stringstream` object. The `stringstream` helps us build a neatly formatted string object that can then be written to the screen, passed into or out of a function, or written to a data file. Another thing that we'll see soon is that we can turn numbers into strings using this handy tool.

When you need to write your program data into a string in a formatted manner, you'll need to create a `stringstream` object and place your data into that object using the “`<<`” operator. (It is exactly the same thing that we do with `cout`.) Then we use the `str()` function to assign the contents into a string. In Program 4-11 we write numeric data into `stringstream` object, and then assign it into a string. You can use all of the `ios` formatting flags that we have for `cout` with a `stringstream` object. Isn't that great?

Program 4-11

```
1 // This program demonstrates how to use a
2 //stringstream object to create a nicely formatted string.
3
4 #include <iostream>           //for cout
5 #include <string>            //for strings
6 #include <sstream>           //for stringstream
7
8 using namespace std;
9
10 int main()
11 {
12
13     cout << "\n Welcome to the StringStream Demo program.\n";
14 }
```

```
15 //create a few data variables that we'll format
16 //into a single string
17
18 //PI to 15 places
19 double pi = 3.141592653589793;
20 float dollar = 1.00;
21 int dozen = 12;
22
23 string text;
24
25 //we want to have the string text contain this:
26 //A dozen is 12, a dollar is $1.00 and
27 //the value of pi to 10 places is 3.1415926536.
28
29 //Create a stringstream object.
30 stringstream ss;
31
32 //Now using our usual cout "tools", we can use
33 //these tools with ss.
34
35 ss << " A dozen is " << dozen << ", a dollar is $";
36 ss.setf(ios::fixed);
37 ss.precision(2);
38 ss << dollar << " and \n the value of pi to 10 places is ";
39 ss.precision(10);
40 ss << pi << ".";
41
42 //now we assign the contents of ss into a string
43 //use the str() function.
44
45 text = ss.str();
46 cout << "\n Here is our formatted text string:\n" << text << endl;
47
48 //Add one more piece of data into the formatted string:
49 ss << "\n There are 2 \"+\\"s in C++.";
50
51 text = ss.str();
52 cout << "\n Here is the final string: \n" << text << endl;
53
54 return 0;
55 }
```



Output

Welcome to the *StringStream Demo* program.

Here is our formatted text string:
A dozen is 12, a dollar is \$1.00 and
the value of pi to 10 places is 3.1415926536.

Here is the final string:
A dozen is 12, a dollar is \$1.00 and
the value of pi to 10 places is 3.1415926536.
There are 2 "+"s in C++.

4.8 Summary

Functions are the backbone of C and C++ programs. To lay the groundwork for learning object-oriented programming, a programmer must have a firm grasp of the basic function. The following key ideas were presented in this chapter:

- Functions require two “sets” of statements: a prototype or declaration, and a function definition—which is the actual function code. The first line of the function is named the function header line.
- The function prototype, or declaration, and function header line are almost identical. They contain the return data type and input data type variable, list parameters.
- The call statement is the location where the function is actually invoked (i.e., used or called).
- The call requires only variable names.
- If a value is returned, it should be assigned into a variable.
- Functions can be overloaded, meaning that two or more functions can have the same name but they must have different input parameter lists.
- The function prototype can have default values assigned to the input variables. When you have default values you do not need to have values or variables in the call statements.
- Global variables are declared outside any function and are seen by all functions in the file.
- Local variables are declared inside the function or in the function header line and are only accessible within the function.
- Static variables are local variables that retain their value after the function is exited. When the function is called again, it “remembers” the values in the static variables from the last time it was called.
- Variables are in scope (exist) in a program as long as the code is executing in the {} where the variable was declared.

An Overview of Common Errors Encountered while Writing Functions

As you begin to write your own functions in your C++ programs, you may find yourself facing these common compile and link errors. It is impossible for this text to show you every error you might see, but it is possible to show you the general types of errors and show you where to look so to fix them. Remember the old adage about giving a man a fish feeds him for a day, but if you teach him how to

fish, he can feed himself for a lifetime? This philosophy holds true when trying to teach programming students how to understand compiler and link errors. Here are three common errors and explanations as to their cause.

Compiler Error: Function Does Not Take parameters

The error that states that a function does not take a certain number of parameters is a classic error beginning programmers find, and it is due to a mismatch between a function prototype and its call statement. Look at Figure 4-7. This figure contains a modified portion of Program 4-2, How Old Are You? To generate this error, the function prototype and call statements do not match in the input lists.



```
void Write();           //prototype shows no inputs
Write(name, age);      //the call statement has 2 inputs
void Write(string name, int age) //this is the first line of the function
```

The compiler sees the prototype, which is telling it that the *Write* function has no inputs, but then the compiler sees the call statement—with two input parameters—hence the compiler error. When you see this errors, always check that the prototype and call statements match!

Link Error: Unresolved External: void _decl Write(class string, int age)

This is one of the most common errors beginning programmers encounter. A typical debugging conversation: “I have an error and I don’t understand it.” “What kind of error is it?” “A link error.” “What is the message?” “Oh it has some decl thing in it!” Ah Ha! The answer is always, go look at the prototype and its function header line—chances are these program statements don’t match. Figure 4-8 illustrates this error. In this example, the prototype and call statements match, but the actual function is missing one of the input parameters. Since C++ allows programmers to overload functions, it doesn’t mind that there is a *Write* function with one

prototype:	void Write();
call:	int main()
	{
	.
	.
	Write(name,age);
	.
	.
	}
function	void Write(string name, int age)
	{
	cout<<"\n"<<name<<"is"<<age"years old.";
	}

Compiler sees the prototype with no inputs

Call statement has two inputs
Hence you can get this error.

HOW TO FIX : Make sure prototype, call, and function header match in the input lists!

Figure 4-7

Compiler Error: Function Does Not Take 2 Parameters.

```

void Write(string name, int age); ← Here is the prototype

int main()
{
    .
    .
    .
    Write(name,age); ← Call statement - parameters match prototype
    .
    .
}

// function not written yet
// OR if function didn't match prototype

void Write(string name)
{
    cout<<"\n"<<name;
}

```

Call statement - parameters match prototype
compiles without error but
↓
Linker is looking for the Write function
that matches the prototype

FIX : If you have a prototype and call to a function, make sure the function is written too!

Figure 4-8

Link Error: Unresolved External void _decl Write(string name, int age)

parameter. The problem is that the compiler saw the 2-input prototype and the call statement, so now the linker is looking for the 2-parameter function! It can't find it, therefore you get an unresolved external error.

```

void Write(string name, int age); //prototype shows 2 inputs
Write(name, age); //the call statement has 2 inputs
void Write(string name) //function header line doesn't match the prototype

```

Whenever your linker gives you an unresolved external error on a function, always double check that the prototype, call statement and function header lines match (i.e., have the same input parameters).

Compile Error: Missing Function Header (old-style formal list?)

In the old days, the C language had a different format for its function header line. The function header line had a semi-colon following it. Today's programmer is adept at using the copy/paste feature of the text/code editor, and sometimes copies the function prototypes and pastes them into the code for the function header line. Good practice, because then he is sure his prototype and function parameters match. The only problem is leaving the semi-colon on the function header line generates this classic error. Figure 4-9 shows this problem.

```

void Write(string name, int age); //prototype written correctly here
void Write(string name, int age); //whoops! no ;
{
    cout << "\n Hi" << name << " you are" << age << "years old.";
}

```

Recognizing the old-style header error means you'll have a quick fix while you are compiling your programs.

Figure 4-9

Compiler Error: Missing Function Header (old-style formal list?).

```
void Write(string name, int age); ← Prototype correctly written with ending;

void Write(string name, int age); ← Problem here
{                                     no ; on function
}                                     header line
                                         ↓
                                         Function Definition
                                         FIX : Remove ; from function header line.
```

4.9 Practice!

IsItPrime?

IsItPrime is Program 4-12. It asks the user to enter a positive integer value. The program checks that the value is positive, and then determines if the number is prime.¹ The program flow is illustrated in Figure 4-10, with the numbered arrows indicating the function call order and values that are passed between functions. This program calls the function *AskPosNumber* to get a value from the user. In the *AskPosNumber* function, it calls the function *CheckIt*. This function checks to see if the value is positive (using a boolean return value, true or false). If it is zero or negative, *AskPosNumber* asks the user to re-enter the value. Once a positive integer is returned to main, we call the *IsItPrime* function, which determines if the number is prime. The *IsItPrime* function returns a true or false to main.

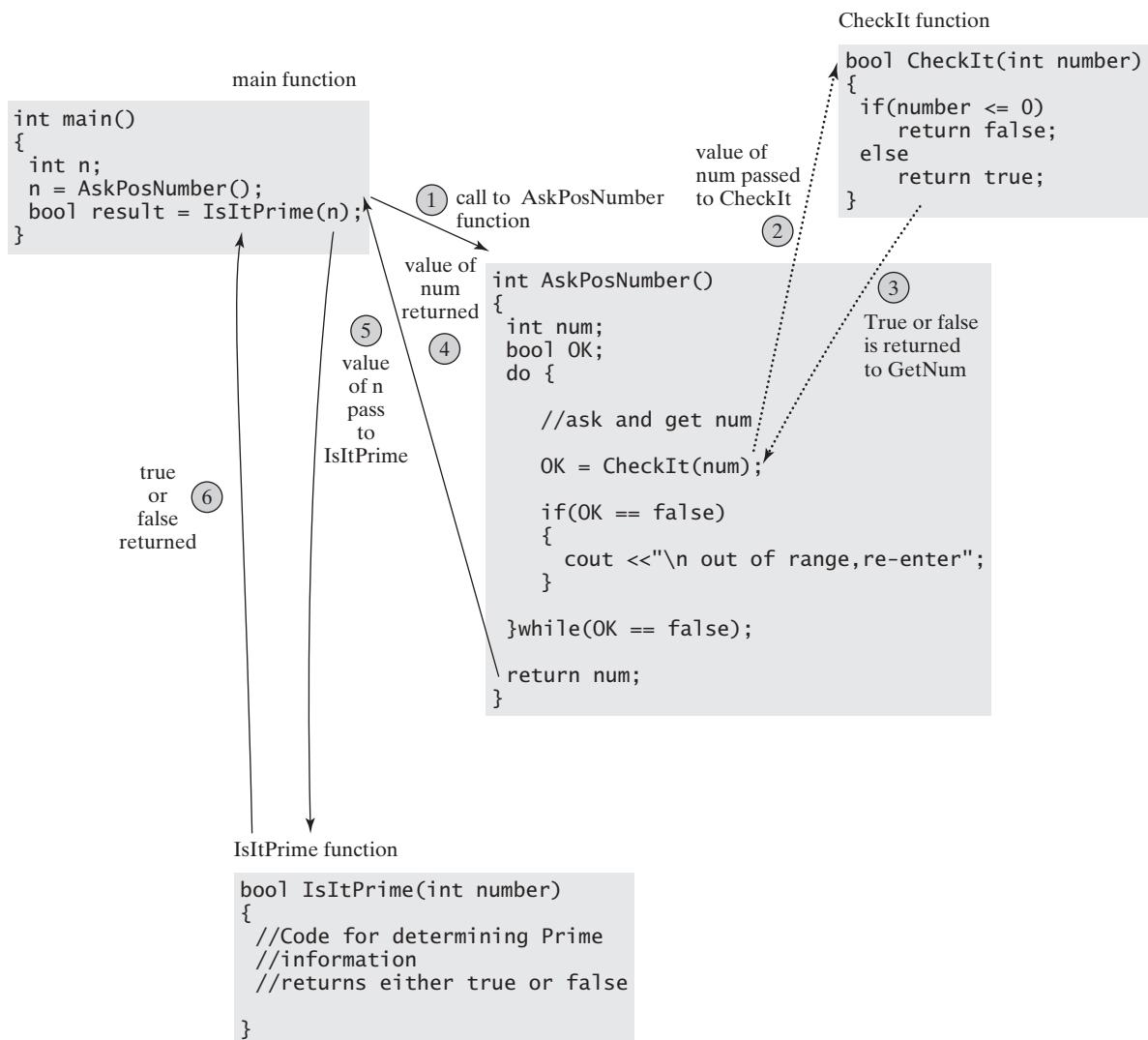
C++ allows the programmer to use the return statement to return only one data item from a function. Therefore, in this program, it is not possible for us to know one of the divisors is for our number. It would be nice if, for example, if the user enters 24, and we are able to see that “24 is not prime and 2 is one divisor.” We could accomplish this task if we used global variables. There are other ways to get two or more values from a function, which we’ll see in the next chapter.

Review Programming Flags One aspect of creating a program with functions involves determining a value convention for returning indicator variables that relay information concerning the function. These indicator variables are known in computer programming as *flags*. A flag can be any type of variable, but integer, character, and boolean are preferred. There are no hard and fast rules for flags, nor is any magic involved when programming with flags. The programmer simply decides on the type of variables and what each value represents. For example, the return code may indicate that an error occurred in the

flags

variables used in programs to indicate certain events or conditions

¹A whole number is considered prime if it is divisible (using whole number divisors) by only 1 and itself. For example, 7 is a prime number because it can be divided only by 1 and 7 evenly. 24 is not a prime number, because 2, 3, 4, 6, and 8 are whole number divisors for 24.

**Figure 4-10**

IsItPrime Program Flow. The circled numbers indicate the order of the function calls and returned values.

function. The value of 1 may mean there was an error, whereas the value of 0 means no error. The *bool* data type in C++ allows the programmer to use the *true* and *false* values. The *CheckIt* function illustrates how to use a *bool* data type for a flag.

A Note on Checking Boolean Values As you are beginning to learn how to check the value of variables, it is important that you write your comparison statement completely. We have been exact in the *if* statements and *while* loops for our

programs. In the code below, the *IsItPrime* program checks the boolean result value with this statement:

```
if(result == true)
{
    cout << "\n\n The value " << number << " is prime. \n\n";
}
else
{
    cout << "\n\n The value " << number << " is NOT prime.";
}
```

As you get more comfortable with the language you see that boolean variables are either true or false, so we do not need to explicitly check "`== true`" or "`== false`". More experienced C++ programmers would write this code like this:

```
if(result)          //boolean value is either true or false
{
    cout << "\n\n The value " << number << " is prime. \n\n";
}
else
{
    cout << "\n\n The value " << number << " is NOT prime.";
}
```

and recognizing the one-line of code within each block would shorten the statements to:

```
if(result)
    cout << "\n\n The value " << number << " is prime. \n\n";
else
    cout << "\n\n The value " << number << " is NOT prime.";
```

The NOT operator (!) negates the value of a boolean variable. Using the NOT operator, we could re-write the last few lines of the *GetPositiveNumber* function like this:

```
OK = CheckIt(number); //true if positive
if( ! OK )
    cout << "\n Value is not positive, please re-enter. ";

}while( ! OK );
```

As we progress through this text, we will use this shortened checking status, and when obvious, may embed function calls inside conditional statements. You should be careful that you don't make your code so obscure that another programmer would have trouble reading your code!

Program 4-12

```
1 //Is It Prime?
2 //Asks the user to enter a positive integer and
3 //determines if the number is prime.
```

```
4
5
6 #include <iostream>
7 #include <string>
8 using namespace std;
9
10 //function declarations
11 int AskPosNumber();
12 bool IsItPrime(int number);
13 bool CheckIt(int number);
14
15 int main()
16 {
17     int n;
18     string answer;
19
20     do
21     {
22         //get the number from the user
23         n = AskPosNumber();
24
25         //now check if it is prime
26         bool result = IsItPrime(n);
27
28         if(result == true)
29         {
30             cout << "\n The value " << n << " is prime.";
31         }
32         else
33         {
34             cout << "\n The value " << n << " is NOT prime.";
35         }
36
37         cout << "\n Do another number? yes/no ";
38         getline(cin,answer);
39     }while(answer == "yes");
40
41     return 0;
42 }
43
44
45 int AskPosNumber()
46 {
47     int num;
48     bool OK;
49
50     do
51     {
```

```
52         cout << "\n Please enter a positive integer ==> ";
53         cin >> num;
54         cin.ignore(); //strip out the Enter key from queue
55         OK = CheckIt(num); //true if positive
56
57         if(OK == false)
58         {
59             cout << "\n Value is not positive, please re-enter. ";
60         }
61     }while(OK == false);
62
63     return num;
64 }
65
66 bool CheckIt(int number)
67 {
68     if(number <= 0)
69         return false;
70     else
71         return true;
72 }
73
74
75 bool IsItPrime(int number)
76 {
77     int remainder, ctr=2;
78
79     //loop from 2 to n-1 and check remainder from modulus
80     //if a number doesn't have a remainder, there is a
81     //value that "goes into" it, therefore not prime
82     while(ctr < number )
83     {
84         remainder = number%ctr;
85         if(remainder == 0)
86         {
87             //ah ha! number has a divisor, not prime
88             return false;
89         }
90         ctr++;
91     }
92
93     //since we divided the number by all values
94     //from 2 to n-1, and didn't have a 0 remainder
95     //the number is prime
96
97     return true;
98 }
```

Output

```
Please enter a positive integer ==> 97
The value 97 is prime.
Do another number? yes/no yes
```

```
Please enter a positive integer ==> 25
The value 25 is NOT prime.
Do another number? yes/no yes
```

```
Please enter a positive integer ==> -16
Value is not positive, please re-enter
Please enter a positive integer ==> 16
The value 16 is NOT prime.
Do another number? yes/no no
```

Revisit PI Calculation

In Chapter 3, Program 3-15, (page 148) we used an infinite series to determine an estimated value of PI. In this practice program, we'll rewrite that program using functions. There are three functions here in Program 4-13, *AskNumTerms*, which asks the user to enter the desired number of terms for our calculation. The *CalculatePI* is passed the number of terms. It uses the *for* loop to accumulate the sum representing the PI value. This value is returned from the function. The third function is the *PrepResultsString*. It uses a *stringstream* object to format the program data into a single string, which is returned to main and printed to the screen. Notice how neat and clean the *main* function is and that the actual work in the program is performed by the various functions. The results show, once again, that the more terms you use for the PI calculation, the better value you obtain from the infinite series.

```
Program 4-13
1 //A program with functions to help us
2 //uses an infinite series to
3 //calculate an estimated value of PI.
4
5 //Here is PI to 20 places
6 //3.14159265358979323846
7
8 #include <iostream>
9 #include <string>
10 #include <iomanip>           //for setprecision
11 #include <sstream>           //for a stringstream object
12 using namespace std;
13
14 //function prototypes
15 int AskNumTerms();
16 double CalculatePI(int nTerms);
```



```
17  string PrepResultsString(double PI, double calcPI, int nTerms);
18
19  int main()
20  {
21      int nTerms;
22
23      //doubles can only store 15 places
24      double PI = 3.141592653589793;
25
26      double calcPI;
27
28      string result, answer;
29
30      cout << "\n PI Calculation Program\n";
31
32      do
33      {
34          nTerms = AskNumTerms();
35          calcPI = CalculatePI(nTerms);
36          result = PrepResultsString(PI,calcPI,nTerms);
37
38          cout << result;
39
40          cout << "\n Do more PI calculations? yes/no ";
41          getline(cin,answer);
42
43      }while(answer == "yes");
44
45      return 0;
46  }
47
48 //Ask the user for number of terms.
49 int AskNumTerms()
50 {
51     int n;
52
53     cout << "\n Enter number of terms for PI calculation:  ";
54     cin >> n;
55     cin.ignore();
56
57     return n;
58 }
59
60 //Calculate value of PI using infinite series equation.
61 double CalculatePI(int nTerms)
62 {
63     //Declare and initialize here
64     double calcPI = 0.0;
```

```
65     double numerator = 4.0, denom = 1.0;
66
67     for(int i = 0; i < nTerms; ++i)
68     {
69         calcPI = calcPI + numerator/denom;
70         denom += 2.0;
71         numerator = -1.0 * numerator;    //flip sign
72     }
73
74     return calcPI;
75 }
76
77 string PrepResultsString(double PI, double calcPI, int nTerms)
78 {
79     stringstream piString;
80
81     piString.setf(ios::fixed);
82
83     piString << "\n Results\n Number of Terms: " << nTerms
84         << setprecision(10) << "    PI = " << PI
85         << "    Calc PI = " << calcPI << endl;
86
87     return piString.str();
88 }
```

Output

PI Calculation Program

```
Enter number of terms for PI calculation: 300
Results
Number of Terms: 300 PI = 3.1415926536 Calc PI = 3.1382593295
Do more PI calculations? yes/no yes
```

```
Enter number of terms for PI calculation: 3000000
Results
Number of Terms: 3000000 PI = 3.1415926536 Calc PI = 3.1415923203
Do more PI calculations? yes/no no
```

Overloaded Functions, Calculate Pay, and Multiple-File Programs

In this program we practice using overloaded functions to calculate various employees' weekly pay. Recall that overloaded functions are functions with the same name but have different input list parameters. We have three *CalcPay* functions in this program; one function for a manager's weekly salary, one for a salesperson's salary, and one for a staff person's salary. The job of all three *CalcPay* functions is to calculate an employee's weekly salary. The formula we use differs according to the type of employee. A manager's weekly pay is 1/52 of her annual salary. A salesperson's weekly pay (in this example) is a base pay plus a percentage of his total

sales for the week. A staff person's pay is based on an hourly rate. Overtime is 1.5 times that rate for every hour over 40.

As we continue to expand our C++ knowledge and experience with each program we write, it is now time to see how the source code is split into many files, not just the single "Driver.cpp" that we've seen so far. The *CalculatePay* program is designed with multiple source code files. That is, this program's function prototypes are located in a header file called *PayFunctions.h*. Their associated function definitions are in the *PayFunctions.cpp* file. The *main* function is in the *PayDriver.cpp* file. Figure 4-11 shows an outline of how the one-file version of the program has been split into three files. The source code below shows the contents of these three files.

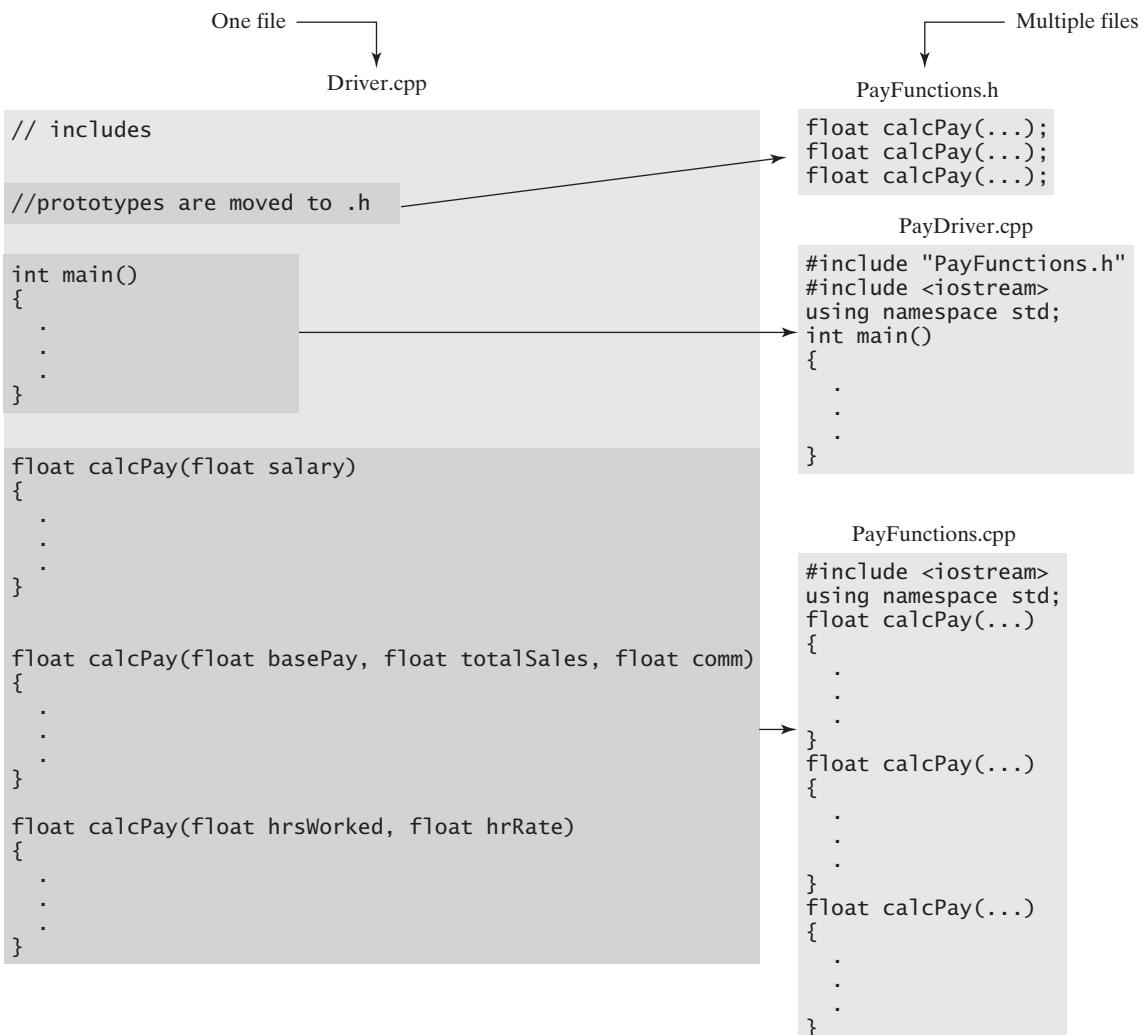


Figure 4-11

The *CalculatePay* program is split into multiple source code files.

Now is an excellent time to read Appendix H, “Multiple File Programs.” It has more examples and further explanations of other multi-file requirements that we’ll need when we begin writing classes. Look over the figure and the source code below, as we are going to be organizing most of our programs in this manner.

Program 4-14

```
1 //File: PayFunctions.h
2 //This file contains the prototypes for the
3 //Weekly Pay Calculation program.
4
5 //We have three overloaded functions named CalcPay.
6
7 //manager: week's pay = salary/52 weeks
8 //salesperson: week's pay = base + comm*totalSales
9 //staff: hrs*salary + 1.5*rate for overtime
10
11 float CalcPay(float salary);
12 float CalcPay(float basePay, float totalSales, float comm);
13 float CalcPay(float hrsWorked, float hrRate);
```

Program 4-14

```
1 //File: PayFunctions.cpp
2 //This file contains the function definitions
3 //for the functions declared in PayFunctions.h
4
5 #include <iostream>
6 #include <sstream>           //for stringstream
7 using namespace std;
8
9 //manager: week's pay = salary/52 weeks
10 float CalcPay(float salary)
11 {
12     float weekPay;
13     weekPay = salary/static_cast<float>(52.0);
14     return weekPay;
15 }
16
17 //salesperson: week's pay = base + comm*totalSales
18 float CalcPay(float basePay, float totalSales, float comm)
19 {
20
21     //assume commission comes is as percentage, i.e. 15%
22     float weekPay;
23     weekPay = basePay + totalSales*comm/100.0;
24     return weekPay;
25 }
```

```
26
27 //staff: hrs*salary + 1.5*rate for overtime
28 float CalcPay(float hrsWorked, float hrRate)
29 {
30     float weekPay;
31     if(hrsWorked <= 40.0)
32     {
33         weekPay = hrsWorked*hrRate;
34     }
35     else
36     {
37         float OT = hrsWorked-40;
38         weekPay = 40.0*hrRate + OT*hrRate*1.5;
39     }
40     return weekPay;
41 }
```

Program 4-14

```
1 //File: PayDriver.cpp
2 //This file contains the main function for
3 //the Weekly Pay Calculation program.
4
5 #include "PayFunctions.h"
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     cout <<"\n This is the Pay Calculation Program.\n"
12         << " We'll calculate three employees' weekly salary. \n";
13
14     //declare variables;
15     float managerPay, salesPay, staffPay;
16
17     //first, a manager making $45000/year
18     managerPay = CalcPay(45000.00);
19
20     //next a salesperson who sold $10000 worth of goods
21     //who has a 20% commission and $200 base pay
22     salesPay = CalcPay(200.00, 10000.00, 20);
23
24     //last, a staff person who earns $12/hr and
25     //worked 45 hours this week
26     staffPay = CalcPay(45,12.00);
27
28     cout.setf(ios::fixed);
29     cout.precision(2);
```

```
30     cout << "\n      Weekly Pay Results"
31         << "\n          Manager: $ " << managerPay
32         << "\n          Salesperson: $ " << salesPay
33         << "\n          Staff person: $ " << staffPay << endl;
34
35     return 0;
36 }
```

Output

This is the Pay Calculation Program.
We'll calculate three employees' weekly salary.

```
Weekly Pay Results
    Manager: $ 865.38
    Salesperson: $ 2200.00
    Staff person: $ 570.00
```

C++ Boat Rental and Multiple-File Programs

Ah summer. Time to pack up the kids, the dogs, the ice chest, swimming suits, and towels and drive out to your favorite lake. Everyone likes to spend time on the water, and some C++ programmers have opened a boat rental business that has a variety of watercraft available for your day at the lake. These programmers-turned-boat business people have written a program to help you select your craft and provide a total cost for the rental. The business has paddle boats, canoes, speed boats, and the ever popular party barge. The program has built in gasoline use and cost for the speed boat and party barge.

The C++ Boat Rental Program is written using multiple files here in Program 4-15. That is, this C++ program is designed using several source code files, instead of just one *.cpp file. The function prototypes are in a header file called BoatFunctions.h. Their associated function definitions are in the BoatFunctions.cpp file. The *main* function is in a file called BoatDriver.cpp. Examine the source code below to see how the functions are organized nicely into the BoatFunctions.h and cpp files.

Program 4-15

```
1  //File:  BoatFunctions.h
2
3  //This file contains the prototypes for the
4  //C++ Boat Rental program.
5
6  #include <iostream>
7  using namespace std;
8
9  string WhatKindOfBoat();
10 int RentalDuration(string boatType);
11 int GasolineReqmt(string boatType);
12 float CalcRentalCost(string boatType, int hours);
13 void WriteGreeting();
```

Program 4-15

```
1 //File: BoatFunctions.cpp
2 //This file contains the function definitions
3 //of for the functions declared in BoatFunctions.h
4
5 #include <iostream>
6 #include <string>
7 using namespace std;
8
9 //display a program greeting
10 void WriteGreeting()
11 {
12     cout << "\n Welcome to the C++ Boat Rental Program"
13             << "\n We have a fine selection of watercraft for your"
14             << " enjoyment. \n";
15 }
16
17 //ask user to select a boat type
18 string WhatKindOfBoat()
19 {
20     int choice;
21
22     cout << "\n Select your boat:"
23             << "\n 1-paddle boat $5/hr"
24             << "\n 2-canoe  $8/hr"
25             << "\n 3-speed boat $30/hr plus gasoline"
26             << "\n 4-party barge $60/hr plus gasoline\n ==> ";
27     cin >> choice;
28     cin.ignore();
29
30     switch(choice)
31     {
32         case 1: return "paddle boat";
33         case 2: return "canoe";
34         case 3: return "speed boat";
35         case 4: return "party barge";
36         default: return "paddle"; //just in case mistyped
37     }
38 }
39
40 //how long to keep the boat?
41 int RentalDuration(string boatType)
42 {
43     int hours;
44
45     cout << "\n How long do you wish to use the " << boatType << "?";
46     cout << "\n Rental options: \n 2 hours \n 4 hours (1/2 day)"
47             << "\n 8 hours (all day) \n ==> ";
```

```
48     cin >> hours;
49     cin.ignore();
50
51 //check to be sure hrs are 2, 4, or 8
52 if(hours == 2 || hours == 4 || hours == 8)
53 {
54     return hours;    //valid rental time
55 }
56 else
57 {
58     //make the user keep the boat all day  ;-)
59     cout << "\n Invalid hours, we're sure you want the" <<
60         boatType << " all day.";
61     return 8;
62 }
63 }
64
65 //if it is party barge or speed boat, set the
66 //gasoline per hour usage rate
67 int GasolineReqmt(string boatType)
68 {
69     int gallonsPerHour;
70     if(boatType == "speed boat")
71         gallonsPerHour = 3;
72     else if(boatType == "party barge")
73         gallonsPerHour = 2;
74
75     cout << "\n The " << boatType << " uses " << gallonsPerHour
76         << " gallons of gas each hour. ";
77
78     return gallonsPerHour;
79 }
80
81 //calculate total rental cost
82 float CalcRentalCost(string boatType, int hours)
83 {
84     float boatCost, gasCost= static_cast<float>(0.0);
85     float gallonsPerHour;
86
87     if(boatType == "speed boat" || boatType == "party barge")
88     {
89         cout << "\n The " << boatType <<
90             " has gasoline $ur-charge $$ ";
91
92         gallonsPerHour = GasolineReqmt(boatType);
93
94         cout << "\n The C++ Gas Center charges only $10/gallon.";
95         gasCost = hours*gallonsPerHour*10;
96     }
```

```
97
98     if(boatType == "paddle boat") boatCost = hours * 5.0;
99     else if(boatType == "canoe") boatCost = hours * 8.0;
100    else if(boatType == "speed boat") boatCost = hours * 30 + gasCost;
101    else //this is the party barge
102        boatCost = hours * 60 + gasCost;
103
104    return boatCost;
105 }
```

Program 4-15

```
1  //File: BoatDriver.cpp
2
3  //This file contains the main function for
4  //the C++ Boat Rental program.
5
6  #include "BoatFunctions.h"
7  #include <iostream>
8  #include <string>
9  #include <iomanip>
10 using namespace std;
11
12 int main()
13 {
14     WriteGreeting();
15
16     string boatType;
17
18     boatType = WhatKindOfBoat();
19     int hours = RentalDuration(boatType);
20
21     float cost = CalcRentalCost(boatType,hours);
22
23     cout << "\n Your " << boatType << " will cost you $"
24         << setprecision(2) << setiosflags(ios::fixed) << cost
25         << " for " << hours << " hours. Have fun! \n\n";
26
27     cout << "\n Would you be interested in renting life jackets? \n";
28
29     return 0;
30 }
```

Output selecting a party barge for 4 hours

Welcome to the C++ Boat Rental Program

We have a fine selection of watercraft for your enjoyment.

Select your boat:

1-paddle boat \$5/hr

2-canoe \$8/hr
3-speed boat \$30/hr plus gasoline
4-party barge \$60/hr plus gasoline
==> 4

How long do you wish to use the party barge?

Rental options:

2 hours
4 hours (1/2 day)
8 hours (all day)
==> 4

The party barge has gasoline sur-charge \$\$

The party barge uses 2 gallons of gas each hour.

The C++ Gas Center charges only \$10/gallon.

Your party barge will cost you \$320.00 for 4 hours. Have fun!

Would you be interested in renting life jackets?

C++ Farm Irrigation Program

Here's an irrigation problem that farmers in the Southwest part of the United States face on a weekly basis. Farmers who rely on irrigation ditches for water are allowed to flood their fields once a week. The irrigation ditches have "checks," which are gates, that the farmer lowers using a wheel-key. When the check is down, the water in the ditch backs up, raising the level of the water. The farmer can then open the gate to his field, allowing the water to flood the field. Depending on the field size and the head of water in the ditch, this activity can take a few hours to all day.

In this last program in Chapter 4, we have written a multi-file program using functions to help the farmer determine the irrigation time and required number of gallons of water. The farmer must know his field size and desired flood depth. We ask for the field size in acres (there are 43,560 square feet in an acre), desired flood depth and the estimated flow rate of water into the field. Converting the volume of water needed to gallons (231 cubic inches in a gallon) gives us total gallons. The flow rate in gallons per minute gives us an easy way to determine the total minutes. The mathematics here is straightforward. Be sure to study the multi-file layout and see how data values are passed between functions. We make use of the stringstream class to format the results string.

Program 4-16

```
1 //File: Functions.h
2
3 //This file contains the prototypes for the
4 //C++ Farm Irrigation program.
5
```

```
6 #include <iostream>
7 #include <string>
8 using namespace std;
9
10 int AskDepthOfWater();
11 float AskIrrigationRate();
12 float AskFieldSize();
13 void WriteGreeting();
14 string CalculateTimeAndGallons(int depth, float rate,
15                                     float acres);
```

Program 4-16

```
1 //File: Functions.cpp
2 //This file contains the function definitions
3 //for the functions declared in Functions.h
4
5 #include <iostream>
6 #include <string>
7 #include <sstream>           //for stringstream
8 using namespace std;
9
10 //display a program greeting
11 void WriteGreeting()
12 {
13     cout << "\n C++ Farm Irrigation Calculation Program"
14     << "\n Given the number of acres, "
15     << " desired depth of water (inches),\n and flow rate"
16     << " from the turn-out gate, \n this program finds the time"
17     << " and gallons needed\n to irrigate the C++ Farm's field. \n";
18 }
19
20 //ask user for size of field in acres
21 float AskFieldSize()
22 {
23     float acres;
24
25     cout << "\n What is the size of your field? (in acres) ==> ";
26     cin >> acres;
27     cin.ignore();
28
29     return acres;
30 }
31
32 //ask user for depth of water desired in the field
33 float AskIrrigationRate()
34 {
35     float gallonsPerMinute;
```

```
36
37     cout << "\n What is the gate's flow rate of water? "
38         << "\n (in gallons per minute) ==> ";
39     cin >> gallonsPerMinute;
40     cin.ignore();
41
42     return gallonsPerMinute;
43 }
44
45
46 //ask user for flow rate from the irrigation gate
47 int AskDepthOfWater()
48 {
49     int depth;
50
51     cout << "\n How deep do you wish to flood your field?"
52         << " (in inches) ==>";
53     cin >> depth;
54     cin.ignore();
55     return depth;
56 }
57
58 //calculate the time and gallon usage
59 string CalculateTimeAndGallons(int depth, float rate, float acres)
60 {
61     int sqFeetInAcre = 43560;
62
63     int totalSqFeet = sqFeetInAcre * acres;
64
65     //convert field size to total square inches
66     //a square foot is 12 x 12 = 144 square inches
67
68     int totalSqInches = totalSqFeet * 144;
69
70     //now multiply by desired inch depth
71     int fieldVolume = totalSqInches * depth;
72
73     //now convert to gallons, divide by 231 cubic in/gal
74     float gallons = fieldVolume/231.0;
75
76     //flow rate is gallons/minute
77     float irrigTime = gallons/rate;
78
79     //now put it all together in a summary string
80     //use a stringstream object to help
81     stringstream ssObj;
82
83     ssObj    << " A " << acres << " acre field "
```

```

84             << "flooded to " << depth << " inches"
85             << "\n requires " << gallons << " total gallons of water.";
86
87     ssObj << "\n\n Irrigation gate flow rate: " << rate
88             << " gallons/minute" << "\n it will take "
89             << irrigTime << " minutes to irrigate this field. ";
90
91     return ssObj.str();
92 }
```

Program 4-16

```

1  //File: WaterDriver.cpp
2
3  //This file contains the main function for
4  //the C++ Farm Irrigation program.
5
6  #include "Functions.h"
7  #include <iostream>
8  #include <string>
9  using namespace std;
10
11 int main()
12 {
13     WriteGreeting();
14
15     string results;
16     float acreage, irrigationRate;
17     int depth;
18
19     depth = AskDepthOfWater();
20     acreage = AskFieldSize();
21
22     irrigationRate = AskIrrigationRate();
23
24     results = CalculateTimeAndGallons(depth, irrigationRate, acreage);
25
26     cout << endl << results << endl;
27
28     return 0;
29 }
```

Output for 1.0 acre field flooded to 3 inches with a flow rate of 250 gpm
C++ Farm Irrigation Calculation Program
 Given the number of acres, desired depth of water (inches),
 and flow rate from the turn-out gate,
 this program finds the time and gallons needed
 to irrigate the C++ Farm's field.

How deep do you wish to flood your field? (in inches) ==> 3

What is the size of your field? (in acres) ==> 1.0

What is the gate's flow rate of water?

(in gallons per minute) ==> 250

A 1 acre field flooded to 3 inches

requires 81462.9 total gallons of water.

Irrigation gate flow rate: 250 gallons/minute

it will take 325.851 minutes to irrigate this field.

REVIEW QUESTIONS AND PROBLEMS

Short Answer

1. What are the three required sets of statements for every function that a programmer writes in C++?
2. Name the three places in a C++ program where variables can be declared.
3. What is the job of the return statement?
4. What is the difference between a prototype statement and a call statement?
5. If your prototype has a void return statement, does this mean you should enter “return 0;” at the bottom of the function? Explain.
6. Why is it unnecessary to pass global variables between functions?
7. When you are working with functions, type mismatch errors are usually what type of error?
8. What is the difference between a static local variable and an automatic local variable?
9. Name two reasons why programmers should be careful when using global variables.
10. What is the difference between a calling function and a called function?
11. Is it possible to have several return statements in a function? Does a return statement have to return a value? Explain both answers.
12. Is an overloaded function the same thing as a called function? Explain.
13. If I declare a string variable named “color” in main, do I need to repeat the declaration in the program functions (assuming the function needs the color variable)?
14. If my program is calling a function that returns an integer value, what do I need to remember to have in the call statement so that I obtain and keep that integer value?

15. Will my program compile, link and run if I have the prototype but not the function body? Will it compile, link and run if I have the function body but not the prototype? Prototype and call, but no function body? Call, function body, but no prototype?
16. Will a C++ program compile and run if it has a call to a function that returns a value but no assignment operator in the call?
17. Why is it silly to have a static declaration (such as static `int count;`) above the `main` function? (That is, why is it silly to have a global, static variable?)
18. Programming flag variables are used in programs for remembering states or conditions. What are the two best data types to use for flags? Give an example of a flag that remembers if a value is negative, zero, or positive.
19. A programmer is writing a function that requires four input values, but most of the time when the function is called, three of the inputs are the same (only changed occasionally). How should the programmer design this function? Design this function so it can be called without always passing the three nearly constant values.
20. A double or a float variable should not be used as programming flag variables. Why is this so?

Reading the Code

Show the screen output from the code examples in Problems 21, 22, and 23.

21. What output will be generated from the following source code?

C Code	Output
<pre>#include <iostream> using namespace std; void Red(); int main() { int i = 1; while(i < 4) { Red(); ++i; } return 0; } void Red() { cout << "\nRed is a great color."; }</pre>	

22. What output will be generated from the following source code?

C Code	Output
#include <iostream> #include <cmath> using namespace std; int SquareIt(float x); int main() { int i; float x = -3.7755; i = SquareIt(x); cout << "The float was" << x; cout << "\nIt was changed to" << i; return 0; } int SquareIt(float x) { int r = static_cast<int>(x*x); return r; }	

23. What output will be generated from the following source code?

C Code	Output
#include <iostream> using namespace std; void SayHello(int n); int main() { int i, j, k; for(i = 0; i < 10; ++i) { if(i%2 == 0) SayHello(i); } return 0; } void SayHello(int n) { cout << "\n Hi there! n =" << n; }	

Debugging Problems

The source code examples in Problems 24 to 26 are incomplete programs. Identify the compiler errors and state what is needed to eliminate the error(s).

24.

```
#include <iostream>
using namespace std;
void Func1();
int Func2(float);
int main
{
    Func2();
    n = FUNC1();
}
```

25.

```
include <iostream>
Int HowOldAreYou(void);
int main
{
    int age;
    age = HowOldAreYou(void);
}
```

26.

```
#include <iostream>
string AskForTime();
int main
{
    string thisTime;
    AskForTime();
}
```

Programming Problems

For all of the following programming problems, have the program write your name and program title to the screen one time. Incorporate a “play again” loop so that the user can repeat the program. Have the program ask if the user wishes to go again utilizing a string for the “yes” or “no” response. When the user is finished “playing,” have the program write out a goodbye message to the screen. Note: do not “trap” the user in a loop forcing him to enter “valid” data! For these problems, if you obtain invalid data, write an error and drop to the “go again” code.

If you are comfortable building multi-file programs, do so.

- 27.** Write a program that uses the random number generator functions to generate a random sequence of letters from A to Z. The *GenerateALetter* function should return a letter based on the ASCII table; that is, A = 65 and Z = 90. In main, above the “do again” loop, ask the user to enter a seed value and call *rand()* passing it the seed value. Write out fifteen randomly-generated characters on one line (separated by commas). *Note:* You will need to place the call to *GenerateALetter* inside a *for* loop and write one letter at a time.

28. Let's rewrite the Mortgage Calculator program in Chapter 2, Problem 46 (page 96) using functions! You will need to write three *Ask* functions, one for the interest rate, one for the principal and one for the number of years for the loan. Pass these values to the *MortCalc* function. This function determines the monthly payment value. Instead of returning a double, return a string that has a description of loan information. The string should have the principal of the loan, interest rate, years, monthly payment and total interest paid. Be sure dollar values are formatted to two decimal places and show the \$ sign. Call all functions from main and write the resultant string to the screen from main.
29. Write a C++ program that encodes the user's string so that he can send a "secret message" to his friends. In order to encode the string, we will use an integer offset value that is added to each letter's ASCII (decimal) value thus making it a new ASCII character. The letter "A" is ASCII 65. If we added 10 to it, it becomes ASCII 75, which is the letter K. For example, if our message was MEET ME AT 8, adding 10 to each letter results in the message WOO^*WO*K^*B.

You'll need to write a function named *AskForString* that obtains the user's string. Request the user to use capital letters, not lowercase. Numbers are fine too. The *main* function generates the encoding integer (also known as the encryption key), which is a random number between 1 and 35. Pass it and the string to the *Encoder* function. It returns the encoded string. We need a decoder function too. It is passed the key and the encoded message—which returns the original message in a string. Write the original, encoded and decoded messages and key value to the screen from main. (Hint: to encode/decode you'll need to pull each character out of the string, adjust it, and then use a *stringstream* object to obtain the new string.)

Extra challenge: If you have explored reading and writing files, you can write two programs, one for encoding and one for decoding. The encoder asks the user for the message string, generates the key, and writes the encoded message to one file and key to another. The decoder asks for the encoded message and key files, decodes the message and writes it to the screen.

30. We're going to revisit the Fruit Finder program from Chapter 3, Problem 47 (page 174). Write a C++ program that calls a function named *AskForFruitString*. It asks the user to enter a sentence containing some fruit information. Pass the string to the *FruitFinder* function. This function searches the sentence for any one of the seven fruits, replaces it with "Brussels sprouts," and returns the string. If the fruit isn't found, the function returns a string reporting that the user must not like fruit. From main write the two strings (original and new string from the function) to the screen.

In the *FruitFinder* function you will need to have local copies of the fruit vector and load it with the seven fruit strings. This is not the most efficient way to code, i.e., refilling the vector each time the function is called, but it will work for us this time. (Once we learn how to use references or write classes, we'll write more efficient code.)

31. Write a program that sets up three overloaded functions called *GetRandomNum*. These functions return a random number that is generated with the *rand()* function. You will need to use the modulus function and some

arithmetic to obtain the desired results. These are the three different prototypes for the *GetRandomNum* function:

1. Pass in a positive integer m and the function returns an integer value between 0 and $m - 1$.

```
int GetRandomNum(int m);
```

2. Pass in two integers (assume i and j are zero or positive and $i < j$) and receive a random integer within and including i and j .

```
int GetRandomNum(int i, int j);
```

3. The version of this function that takes no inputs returns a random number between 0.000 and 1.000. Assume that three digits of precision are necessary.

```
double GetRandomNum();
```

Your *main* function should have a *for* loop that executes fifteen times. Inside the loop call each function and print the resultant values, all three in one line. Align your columns of values so the output looks like a table. (You'll need to write the table header before you call the loop.) In order to test your functions your program should have 100 as an input for the first version, and -100 to $+100$ for the second. Write your decimal values showing three digits of accuracy.

32. Write a program that has a function called *WriteHellos* that uses a default input parameter list in the prototype. The default parameters include an integer for the number of hellos on the line and an integer for the number of lines. The default function result shows five hellos on three lines. This call: *WriteHellos()*; results in output like this:

```
hello hello hello hello hello  
hello hello hello hello hello  
hello hello hello hello hello
```

If either input value is zero, write “No Hellos for You.” Your *main* function should call this function four times—once using the default values, once so that you see twelve hellos on three lines, once so that you see five hellos on eleven lines and once showing the “No Hellos for You” message.

33. Chapter 4’s Is It Prime program (4–12, page 218) has the code that determines if an integer value is a prime number. Borrow the *IsItPrime* function and use it to find the user-requested number of prime numbers. For this program, write a *ShowPrimes* function that is passed an integer. This value is the user’s requested number of primes. (For example, the user may wish to see the first 25 prime numbers, beginning at 1.) Inside this function is a *while* loop which continues to call the *IsItPrime*, until it has found that number of prime numbers. The *ShowPrimes* function writes them neatly in the output window, showing rows of 10 primes. Your *main* function should call a *HowManyPrimes* function, and then the *ShowPrimes*.

34. Write a program that models a number guessing game. The way the game is played is this—it asks the user to select a range of numbers from zero to a positive integer. It uses a random number generator to generate a “magic” number within this range. The program should then give the user five chances to guess the value of the magic number. When the user enters the guess value, the program reports if the number is too high or too low. If the user guesses

the number, the program should print a congratulatory message and report the number of tries it took to guess the number. If the user does not guess the number in five tries, the program tells the user the value of the magic number.

We're going to use functions to build this program. There should be an *AskRange* function that obtains the positive number that is the top value for the guessing range. The *GetMagicNumber* is passed the top range and returns a value between 0 and this number. The range and magic number are passed to the *Guess* function. The logic for the five guesses resides in this function including the congratulatory message or showing the number if the user is out of guesses. (Hint: make use of the ability to return from a function in multiple locations! It'll make the logic much easier to code.)

35. In Chapter 3, we practiced writing for loops by drawing hollow or solid rectangles of characters. (See problem 44, page 172.) The user was asked to enter the number of rows, number of columns, a character and select whether hollow or solid. Let's rewrite this as a set of overloaded functions named *DrawRect*. Write one *DrawRect* that is passed the number of rows, columns, character to draw and a flag of some sort indicating hollow or solid. To make this interesting, set these default values in the prototype: a hollow 5 rows, 25 columns, and asterisk (*) character. (You determine the best order for the inputs.) The second version of the *DrawRect* function centers a string of text within the rectangle. This version does not receive the hollow/solid flag, as it is always hollow. It is passed the number of rows, columns, draw character and string of text. The default string value is no text, i.e., blank (" "). The text is centered in the middle of the rectangle. The rectangle must be at least four characters wider than the requested text. If the text is longer than the requested width, then increase the number of columns to be four more than the length of the string. If you have an even number of rows, center the text in the upper portion of the rectangle (see sample).

Request: 6 rows by 20 columns, center the phrase I love C++ (The text fits within the rectangle, so the rectangle is 6 × 20.)

```
&&&&&&&&&&&&&&&&&  
& &  
& I love C++ &  
& &  
& &  
&&&&&&&&&&&&&&
```

Request: 5 rows by 25 columns, center the phrase "Dogs are wonderful company." (Phrase contains 27 chars, thus we make the rectangle width 31 chars.)

```
&&&&&&&&&&&&&&&&&&&&&&  
& &  
& Dogs are wonderful company. &  
& &  
&&&&&&&&&&&&&&&&&&&&
```

Write a *main* function that calls each function three times, illustrating a different number of parameters each time. Your *main* function can either ask the user for values or you can hardcode them. If you hardcode the values, make sure to write descriptive messages to the user so he knows what your program is doing.

36. A palindrome is a word or a textual phrase that contains the same sequence of letters forward and backward (spaces are ignored). For example, HANNAH is a palindrome, as is RADAR, NURSES RUN, and I PREFER PI. Write a function *AskForString* that receives a string of text. The function *IsPalindrome* is passed the string and returns a bool value, true indicating the string is indeed the same sequence of characters, forward and backward. Have main call both functions, and write the string and state whether it is a palindrome. Note: spaces are ignored.
37. Let's rewrite and expand the C++ Golf and Spa Resort lake program in Chapter 2 (Problem 54) using functions. Read this program statement (page 99) for program background. Because the golf course has several lakes, ask him to enter the name of the lake in an *AskForLakeName* function, the lake's surface area (acres) in *AskForLakeSize*. The *AskForLakeDrop* function asks the groundskeeper to enter the number of inches that the lake has dropped over the week. Call these three functions, returning their values to main.

The C++ Golf and Spa Resort owns several of the high quality, C++ Water Pumps, which pump 10,000 gallons of water in an hour. Pass the dropped inches, lake name, and lake size to the *CalcPumpTime* function that calculates the required gallons and determines the time the groundskeeper must run the water pump to refill the lake. Write the lake name, size (in acres), drop (in inches) and required hours and minutes for the pump to the screen from this function. (Design issue: You may write a *CalcGallons* function in which the *CalcPumpTime* uses. It could be passed the drop-inches and lake size, and determines and returns the required number of gallons.)

38. We've written a few programs for The C++ Pond Pump Manufacturers for determining pumps and skimmers. Another popular product is the C++ Fish Shower. This handy little device can be adapted to your C++ Pond Pump and it provides your pond fish a shower that they may use any time. The Fish Shower sprays a cone of water into the air causing rain-like effect on the surface of your pond, as well as producing a soothing splash-splashy sound. (You probably think I am making this up, don't you?)

You should write a function named *AskFishShowerTypes*, which presents the user with five Fish Shower Models. (It should return a string.) Two examples of fish showers may include the "Shebunkin Sprayer," and "Koi Rain Head." (Look up Pond Fish on the web and come up with three other appropriately named shower types.). You should also make up a recommended type-to-volume specification for your shower types. For example, the Shebunkin Sprayer is optimum for small ponds between 200 and 400 gallons, whereas the Kio Rain Head is an excellent choice for ponds between 300 and 1000 gallons. (You'll use this in the *Write* function.)

Next you must determine the number of gallons in the user's pond. Call *CalcPondVol*, which asks pond shape (circular or rectangular) and it determines the pond volume. You may assume the pond has a constant depth. (See Problem 50 in Chapter 2.) Lastly, pass the user's desired shower type and volume into the *WriteShower* function. It reports if the selected shower type is appropriate for the user's pond. If it is not, it suggests the perfect fish shower. It is important that the pond be fitted with the correct shower for both the pond owner and pond fish's enjoyment. Be sure your five showers are sized for ponds from 200 to 10000

gallons and that the types overlap at the min/max ranges. Have main call all functions. Hint: can vectors make your programming task easier?

39. The C++ Water and Ice Company has another problem for us to solve. When water freezes into ice it expands in volume but still weighs the same. The same volume of liquid water expands to fill a volume approximately ten percent larger. (If you had 100 ounces of liquid water and froze it, the corresponding ice volume is equivalent to 110 fluid ounces.)

The company produces cubes of ice that it sells in bags. To make the ice, square trays have square compartments that are filled with water. For example, trays may contain 100×100 cubes, each cube is intended to be one square inch. The ice trays are filled with water and frozen. This program allows the ice makers to enter ice cube tray sizes and cube size. We determine how much water (in gallons) is required to fill that tray so that the resultant cubes have expanded to completely fill the cube but doesn't expand over the lip of the tray! The program functions are *AskForTraySize* (asks for the number of cubes on the tray), *AskForCubeSize* (asks for single dimension for the cubes side in inches) and *CalcGallons* that is given the tray and cube sizes. The function returns the gallons needed. Remember to use the water to ice expansion information to obtain accurate results!

40. Our C++ programmer built a cabin in the mountains that she visits when she needs to get away from the computer. She placed a black, spherical tank on the hill above her house to provide water for dishes, bathing, etc. (Solar heating!) The water is captured on the roof during rainstorms (she has a very tall cabin) and the water drains into the tank. Use a function to ask the user to enter the radius of the tank. Pass this value to *CalculateTankVolume* to obtain total number of gallons in the tank. Ask the user for the rate of flow she experiences using this gravity fed system by calling *AskFlowRate*. (We'll assume it is a constant flow rated in gallons per hour.) Pass the total gallons and flow rate to the *Usage* function that determines the total number of hours and minutes of water available when the tank is full. Write the water availability, total gallons and flow rate from usage.

41. Wouldn't it be nice to have a helper in the kitchen? The C++ Cook's Helper program provides our chef with an easy way to convert those pesky volumetric measurements. Recall that a gallon is four quarts, a quart is two pints, a pint is two cups. There are eight ounces in a cup. An ounce is two tablespoons and a tablespoon is three teaspoons. As a starting point, write a series of functions that make the conversions listed above (going both ways). That is, write a *GallonsToQuarts*, which is passed gallons and returns quarts, and a *QuartsToGallons*, which is passed quarts and returns gallons. You'll have twelve functions. Be sure to use descriptive names for each converter function.

Write a C++ Cook's Helper program that provides the user with a menu of five possible conversions she might need in the kitchen. The menu accesses custom built conversion functions. You need to decide what conversions would be most useful. Here's one, convert *QuartsToOunces* that is passed quarts and returns ounces. (For example, in converting *QuartsToOunces*, you'd pass the quart value to *QuartsToPints*, and the pint value to *PintsToCups* and cup value to *CupsToOunces*.) Your main should present your menu and allow the user to enter a choice. Obtain the input value, call the menu's associated function. Then write both input and result to the screen.



5

KEY TERMS AND CONCEPTS

assigning address into a pointer
hex address
hexadecimal notation
memory
memory efficient programming
pointer
pointer declaration
references
stack
call-by-reference with pointers
call-by-reference with reference parameters

KEYWORDS AND OPERATORS

address operator, &
indirection operator, *
reference operator, &
sizeof

Functions

Part 2: Variable Addresses, Pointers, and References

CHAPTER OBJECTIVES

Demonstrate how data variables and computer memory are related.

Introduce the address operator, which returns a variable's hex address.

Illustrate the concept of a pointer, which is a variable that holds another variable's address.

Show how the pointer can be used to access another variable's contents with the indirection operator.

Introduce the C++ reference variable.

Present an overview of how pointers and references are used in C++ programs and why they are important.

Explain when it is necessary to use pointers and references with functions.

Show the correct use of pointers and references in functions.



We're Good Up To Here.

You have been introduced to a lot of C++ topics in the past four chapters. You are becoming better at using the C++ string, stringstream, and vector classes, making objects and using them to call the class functions. You are writing your own functions, and even splitting your source code into multiple files. Excellent! However, we are missing one tool in our function-writing toolkit. What could that be, you wonder? Here is a question for you. What if you need to return more than one data item from a function? For example, you wish to write a function that asks the user to input the dimensions of a box, length, width, and height. Wouldn't it be better to just have one function that obtains these three items instead of writing three functions, or (horror!) using global variables?

This chapter fills in the missing pieces for us regarding "returning" several items from a function. We have to delve deeper into how variables are stored in memory in order to understand the two new tools we're going to be using. We'll learn the older, C-style technique that uses "pointers," and then see how to work with C++ references. Programmers may argue that we should always use references because they are easier, but if you progress to developing more advanced C++ problems or writing graphic user interface programs (such as wxWidgets code) you'll need to be an expert with pointers and references. We'll cover both here so you'll be ready for anything in the future.

5.1

Data Variables and Memory

We need to cover some background material here to fully understand the core topics in this chapter. When data variables are declared in a program (such as `int x;`), physical memory in the random access memory (RAM) is reserved for the variable. The number of memory bytes that are actually used depends on the data type of the variable. Table 5-1 (a duplication of Table 2-6 in Chapter 2, page 23) lists all the data types, the bytes reserved, and ranges as dictated by the ISO C++ Standard. (We'll need to have this table handy as we progress through this chapter.)

TABLE 5-1

Data Types Defined by the ANSI/ISO C Standard

Keyword	Typical Bytes of Memory	Precision Range
<i>char</i>	1	–128 to 127
<i>unsigned char</i>	1	0 to 255
<i>signed char</i>	1	–128 to 127
<i>int</i>	4	–2,147,483,648 to 2,147,483,647
<i>short int</i>	2	–32,768 to 32,767
<i>unsigned short int</i>	2	0 to 65,535
<i>unsigned int</i>	4	0 to 4,294,967,295
<i>long int</i>	4	–2,147,483,648 to 2,147,483,647
<i>unsigned long int</i>	4	0 to 4,294,967,295
<i>float</i>	4	6 digits, i.e., 0.xxxxxxx
<i>double</i>	8	3.4 E ± 38 (7 digits in Visual C++)
<i>long double</i>	10	10 digits, i.e., 0xxxxxxxxx
		1.7 E ± 308 (15 digits in Visual C++)
		10 digits, i.e., 0xxxxxxxxx
		1.2 E ± 4932 (19 digits in Visual C++)

sizeof Operator

sizeof

an operator that returns the number of bytes reserved for the variable or for data type

C++ provides the *sizeof* operator, which returns the number of bytes reserved for either a data type or a variable. The form for the *sizeof* operator is:

```
sizeof variable_name; //for a variable value () optional, but convenient
sizeof (data_type); //for a data type (must be in parentheses)
```

In the short program of Program 5-1, we use the *sizeof* operator with the variable name to determine the bytes of memory reserved for five popular data types. Compare the results with the values shown in Table 5-1.

Program 5-1

```
1 //A program that demonstrates the sizeof operator.
2
3 #include <iostream>
4 using namespace std;
5
6 int main()
7 {
8     char c;
9     double d;
10    float f;
11    int i;
12    long l;
13
14    //Use sizeof with data types
```

```

15     cout << "\n The sizeof Demonstration Program.";
16     cout << "\n Number of bytes reserved in RAM for:"
17         << "\n     char   " << sizeof(c)
18         << "\n     double " << sizeof(d)
19         << "\n     float  " << sizeof(f)
20         << "\n     int    " << sizeof(i)
21         << "\n     long   " << sizeof(l) << endl;
22
23     return 0;
24 }
```

Output

The sizeof Demonstration Program.

Number of bytes reserved in RAM for:

```

char 1
double 8
float 4
int 4
long 4.
```

Reserving Memory

In the following three declaration statements, memory space is reserved for these six variables—4 bytes of memory for each *float* and *int*, and 8 bytes for each *double*.

```

float x,y;           //4 bytes reserved for each float and int
int i,j;
double q,r;          //8 bytes reserved for each double
```

The computer reserves actual memory locations for program variables. If the variables are declared inside a function (i.e., local variable), they are placed on the **stack**. The stack is a region of memory reserved for program variables. The first variable that is declared is placed at the far end of the stack. Data variables are stacked into memory, and the first variable—because it is located at the end of the stack—has the highest address. We will not worry about the stack, but when the memory locations are examined, we see that the last declared variable (in this case “*r*”) has the lowest memory address. Figure 5-1 illustrates how the six variables in the declaration statements above are reserved in memory. The boxes in Figure 5-1 represent the memory for each variable.

stack

the portion of the computer memory where local program variables are stored

hexadecimal notation

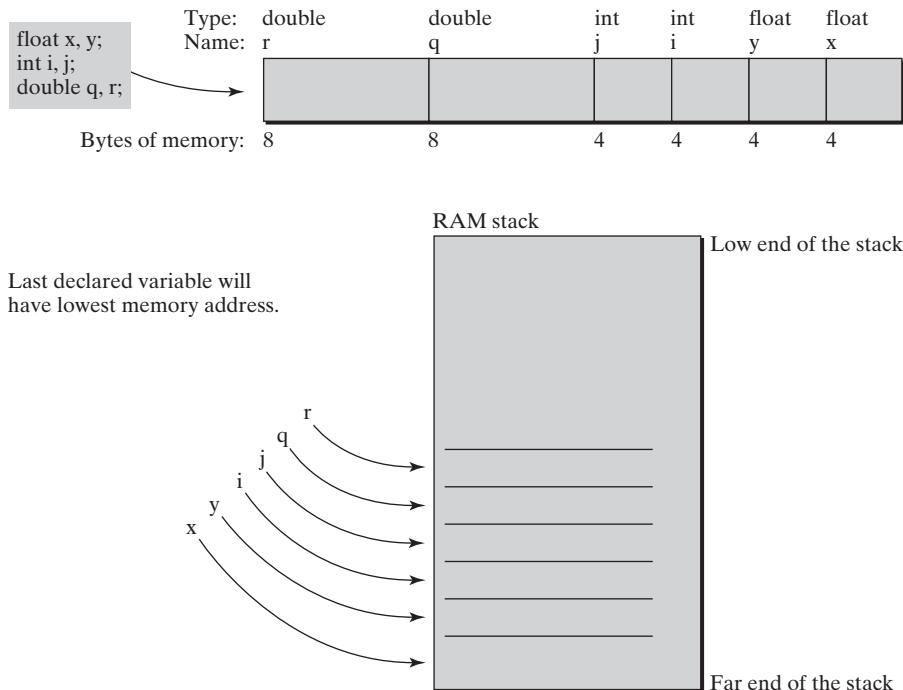
the notation for showing how computer memory is referenced

hex address

the address of a memory location of a variable

Computer Memory and Hex

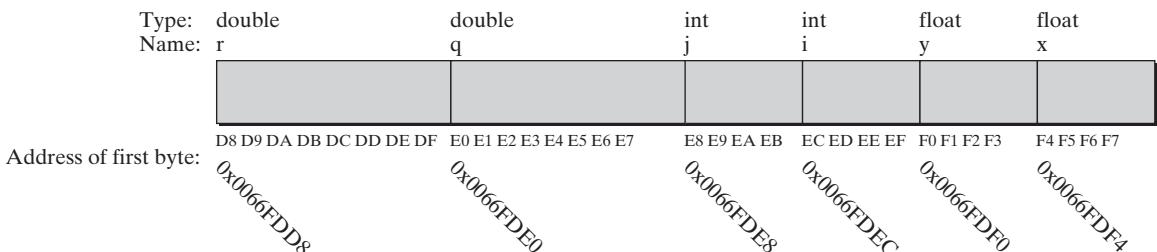
Computer memory is addressed by using **hexadecimal notation**. (For a discussion of hexadecimal notation, see Appendix E, “Bits, Bytes, Memory, and Hexadecimal Notation.”) In a 32-bit environment, the addresses are 4 bytes long. A memory location might have the **hex address** of 0x0066FDF0.

**Figure 5-1**

Memory reserved for local data variables is placed on the stack.

In Figure 5-2, the six variables in the declaration statements above are shown with memory addresses. This diagram assumes that the memory address for the first byte of variable “r” is located at 0x0066FDD8. Each of the six variable addresses is shown.

If we assign values to the six variables, as shown in the code below, each value is stored in memory. In Figure 5-3 the variable values are written in the boxes. Floating point variables have six digits of precision, doubles have at least ten, and integers are whole numbers.

**Figure 5-2**

Data variables and their hexadecimal addresses.

	Type: double Name: r	Type: double Name: q	Type: int Name: j	Type: int Name: i	Type: float Name: y	Type: float Name: x
Value:	6.00000000000000	5.00000000000000	4	3	2.000000	1.000000
Address of first byte:	0x0066FDD8	0x0066FDE0	0x0066FDE8	0x0066FDEC	0x0066FDFO	0x0066FDF4

Figure 5-3

Values are assigned into variables and stored in memory.

```
//assign values into the variables
x = static_cast<float>(1.0);           //x and y are floats
y = static_cast<float>(2.0);           //
i = 3;                                 //i and j are ints
j = 4;                                 //q and r are doubles
q = 5.0;
r = 6.0;
```

5.2

Address Operator: &

Data types, such as floating point, integer, and char, each hold (or contain) a type of data such as numeric or character. Each variable declared in C++ has a data type, name, value, and address. To access the address, C++ provides the **address operator**, **&**. When used with a data variable, the address operator gives the address in memory of the data variable. The sample program in Program 5-2 shows the address operator used with the six variables to write out the memory addresses for each variable. Refer again to Figure 5-3.

address operator, &
 returns the hex
 address of the
 memory location
 for a variable

Program 5-2

```
1 //A program that shows how to use the
2 //address operator to obtain variable memory addresses.
3
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     float x = static_cast<float>(1.0);
10    float y = static_cast<float>(2.0);
11    int i = 3, j = 4;
12    double q = 5.0, r = 6.0;
```

```

13
14     cout.setf(ios::fixed);
15     cout.precision(7);
16
17     cout << "\n A program that shows variable values and addresses.";
18     cout << "\n Value of x = " << x << " Address = " << &x;
19     cout << "\n Value of y = " << y << " Address = " << &y;
20     cout << "\n Value of i = " << i << " Address = " << &i;
21     cout << "\n Value of j = " << j << " Address = " << &j;
22
23     cout.precision(10);
24     cout << "\n Value of q = " << q << " Address = " << &q;
25     cout << "\n Value of r = " << r << " Address = " << &r << endl;
26
27     return 0;
28 }
```

Output

```

Value of x = 1.000000 Address = 0x0066FDF4
Value of y = 2.000000 Address = 0x0066FDF0
Value of i = 3 Address = 0x0066FDEC
Value of j = 4 Address = 0x0066FDE8
Value of q = 5.000000000 Address = 0x0066FDE0
Value of r = 6.000000000 Address = 0x0066FDD8
```

Five Properties for C++ Variables

Whenever you declare a variable in C++, there are five properties associated with it. They are:

1. data type, what sort of data will it hold
2. value, what is the actual data value
3. name, all variables have a name that you, the programmer uses to refer to it
4. memory address, each variable has its own address in the computer's RAM, and
5. how much memory is reserved for the variable? A specific amount of memory is reserved for each type of data.

We do not need to worry about where the variables are stored, but we do need to be aware that each variable is at an exact location in memory, and that location is a hex address. The address operator (`&`), when used with a variable name, returns the address of the variable. We saw how the `&` operator works in the previous program. In Program 5-3, we declare one integer, one double and one string, and assign values to them. Using the address and `sizeof` operators, we are able to see the five properties listed above for these three C++ data items.

Program 5-3

```
1 //This program shows the five properties of
2 //these three C++ data items.
3 //A variable's memory address (in hex notation)
4 //is obtained by using the address operator &.
5
6 //The sizeof operator gives us the number of bytes
7 //that are reserved for C++ variables.
8
9 #include <iostream>
10 #include <string>
11 #include <iomanip>           //for setw
12 using namespace std;
13
14 int main()
15 {
16     //declare and assign variables
17     int n = 42;
18     double pi = 3.14159;
19     string s = "I love C++";
20
21     //write out variable's type, name, address, and bytes
22     cout << "\n A program that shows the & and sizeof operators.\n";
23
24     cout << "\n Data Type    Name    Value      Address    Bytes";
25
26     cout << "\n    int      n " << setw(8) << n << setw(14)
27             << &n << setw(6) << sizeof(n);
28     cout << "\n    double    pi " << setw(9) << pi << setw(12)
29             << &pi << setw(6) << sizeof(pi);
30
31     cout << "\n\n  Class     Name    Value      Address  ";
32     cout << " \n string    s " << setw(14) << s << setw(12)
33             << &s << endl;
34
35     return 0;
36 }
```

Output

A program that shows the & and sizeof operators.

Data Type	Name	Value	Address	Bytes
int	n	42	0012FF70	4
double	pi	3.14159	0012FF68	8

Class	Name	Value	Address
string	s	I love C++	0012FF58

5.3 Pointers

All data types in C++ are designed to hold certain kinds of data. Specific operations are allowed if they are relevant to the data types. For example, an integer variable contain a whole number. The arithmetic and modulus operations are allowed for integer variables. The computer reserves 4 bytes of memory for storing the value for each integer.

pointer

variable that holds the hex address of another variable

A **pointer** is a data type in C++, and it is designed to hold a hexadecimal address, such as 0x0066FDF4. A pointer in a C++ program is meant to hold the address of a specific variable. When the pointer variable contains another variable's address, it is said that the pointer “points to” that variable.

Pointers do not have their own keyword, such as *int*, *float*, or *double*, but pointers have specific pointer declaration statements. The program and the pointer need to know what type of variable-address the pointer contains. A pointer variable is declared by specifying the data type to which it will be pointing. The asterisk operator (*) is used in the declaration statement. Here is the format:

```
data_type *variable_name;
```

An example of a pointer declaration is:

```
float *x_ptr;
```

C++ does not care if the (*) operator is beside the data type or the variable name. This is a valid declaration too:

```
float* x_ptr;
```

You can declare a pointer and have the asterisk between the data type and variable, but this convention is rarely seen in professional code:

```
float * x_ptr;
```

Many naming conventions are used with pointer variables to help the programmer remember which variables are pointers. A naming convention is a general agreement or customary practice used for naming variables, but it is not a rule in C++. Three conventions are listed below:

<code>float *x_ptr;</code>	<code>// use the extension _ptr</code>
<code>float *pX;</code>	<code>// use lowercase p with capital letter</code>
<code>float *p_x;</code>	<code>// use prefix p_</code>

Here we declare two integer variables and two pointers:

```
int count, temp;
int *count_ptr, *temp_ptr;
```

We have named our pointers *count_ptr* and *temp_ptr* so that we can keep track of which pointer variable belongs with which integer. C++ does not automatically assign the correct address into the correct pointer. The address assignment is the job of the programmer. The programmer's tool is the address operator: &.

Pointers and the Address Operator

The address operator (`&`), when used with a variable name, returns the address of the variable. Using the address operator is one way to assign addresses into pointer variables. Program 5-4 illustrates the declaration of variables including pointers and assigning addresses, followed by printing the value and addresses of the variables to the screen. Examine Figure 5-4 and output from this program. The important thing to notice here is that the addresses of the variables are the values in the pointers! (That is, the pointer variable contains a hex address.)

Program 5-4

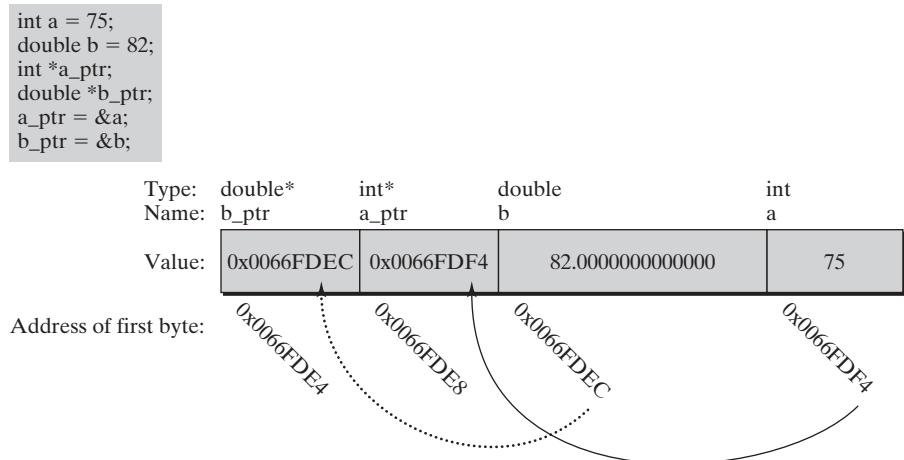
```

1  //Variables, Addresses and Pointers
2  //In this program we declare an int and double,
3  //and pointers, and assign the addresses
4  //into the pointers.
5
6  #include <iostream>
7  #include <iomanip>
8  using namespace std;
9
10 int main()
11 {
12     int a = 75;
13     double b = 82.0;
14     int *a_ptr;
15     double *b_ptr;
16
17     //now we assign the address of vars into pointers
18     a_ptr = &a;
19     b_ptr = &b;
20
21     //write out info
22     cout << "\nVARIABLE      VALUES      ADDRESSES" <<
23         "\n a      " << setw(12) << a << setw(12) << &a <<
24         "\n b      " << setw(12) << b << setw(12) << &b <<
25         "\n a_ptr " << setw(12) << a_ptr << setw(12) << &a_ptr <<
26         "\n b_ptr " << setw(12) << b_ptr << setw(12) << &b_ptr << endl;
27
28     return 0;
29 }
```



Output

VARIABLE	VALUES	ADDRESSES
a	75	0x0066FDF4
b	82	0x0066FDEC
a_ptr	0x0066FDF4	0x0066FDE8
b_ptr	0x0066FDEC	0x0066FDE4

**Figure 5-4**

Each pointer variable is assigned a variable's address.

5.4

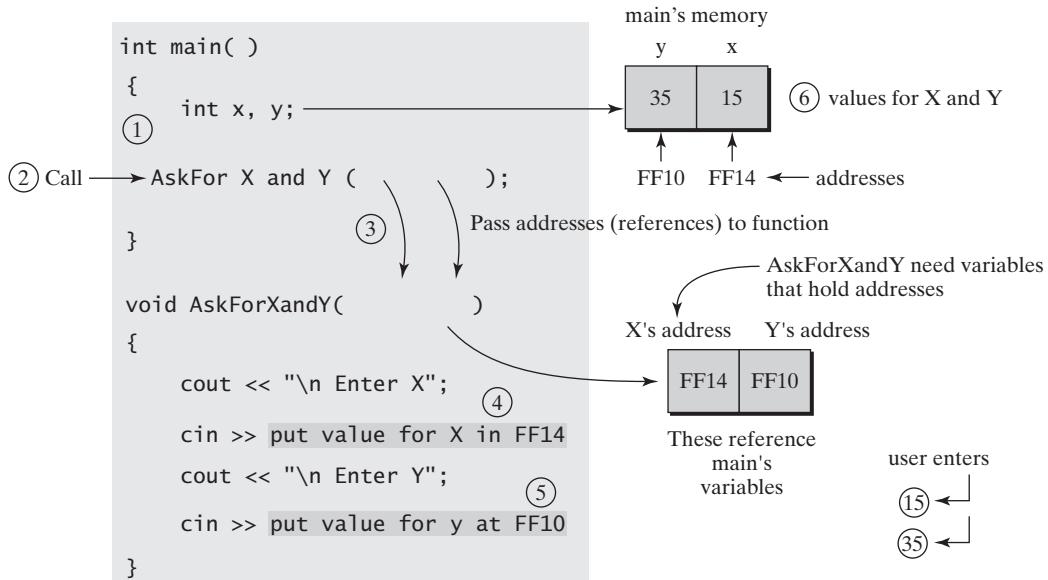
Functions, Pointers, and the Indirection Operator

Big Picture Time

Let's keep in mind what it is we are trying to understand with regards to functions. We are going to see how the C++ programmer can use two different techniques to obtain more than one value from a function. Before we start examining syntax for our two approaches, a simple example and diagram is in order.

Assume that we have short program with two functions, the *main* function and a function called *AskForXandY*. The *AskForXandY* function's job is to ask the user for two integers, *x*, and *y*. The scheme that is employed by the program is this: in *main* we declare our variables, *x* and *y*. We pass the addresses of *x* and *y* to the *AskForXandY* function. When the *AskForXandY* function uses the *cin >>* to read in the values, the values are placed directly into *main*'s memory because we have their memory addresses. Remember, we have given *AskForXandY* address references to *main*'s variables, so when we use *cin*, we have *cin* write the values into *main*'s memory.

Examine Figure 5-5. This figure illustrates the big picture of how we will obtain more than one value from a function. Think of a variable's hex address as a reference to this variable. We'll pass *main*'s *x* and *y* variable addresses to the *AskForXandY* function. The function will use the addresses to place values into the *main*'s memory. C++ gives us two ways "to pass addresses" to a function—but in the big scheme of things, both ways are working in the same manner.

**Figure 5-5**

Big Picture: pass addresses of main's variables to the `AskForXandY` function. `AskForXandY` uses these references to assign values into main's variables.

Pointers and Functions

We are now going to see how pointers, addresses, and a new operator (the **indirection operator**) gives us one approach at solving our problem of returning more than one data item from a function. The C language used this technique, and of course, C++ does too. This technique allows us to perform a **function call by reference using pointers**. We introduce the **indirection operator**, the asterisk (*), which is a pointer's best friend. It works like this. We explicitly pass the address of a variable to a function using the address operator. The function stores the address in a pointer variable. It then uses the indirection operator to access the calling function's variables. The indirection operator, when used with a pointer variable, is telling the program, “Go to the address that I am holding.” The pointer/indirection operator combination can be used either to assign a value where the pointer is pointing, or to get a value where the pointer is pointing. (Now that is a \$50 paragraph if you ask me, but it certainly needs some clarification, don't you think?)

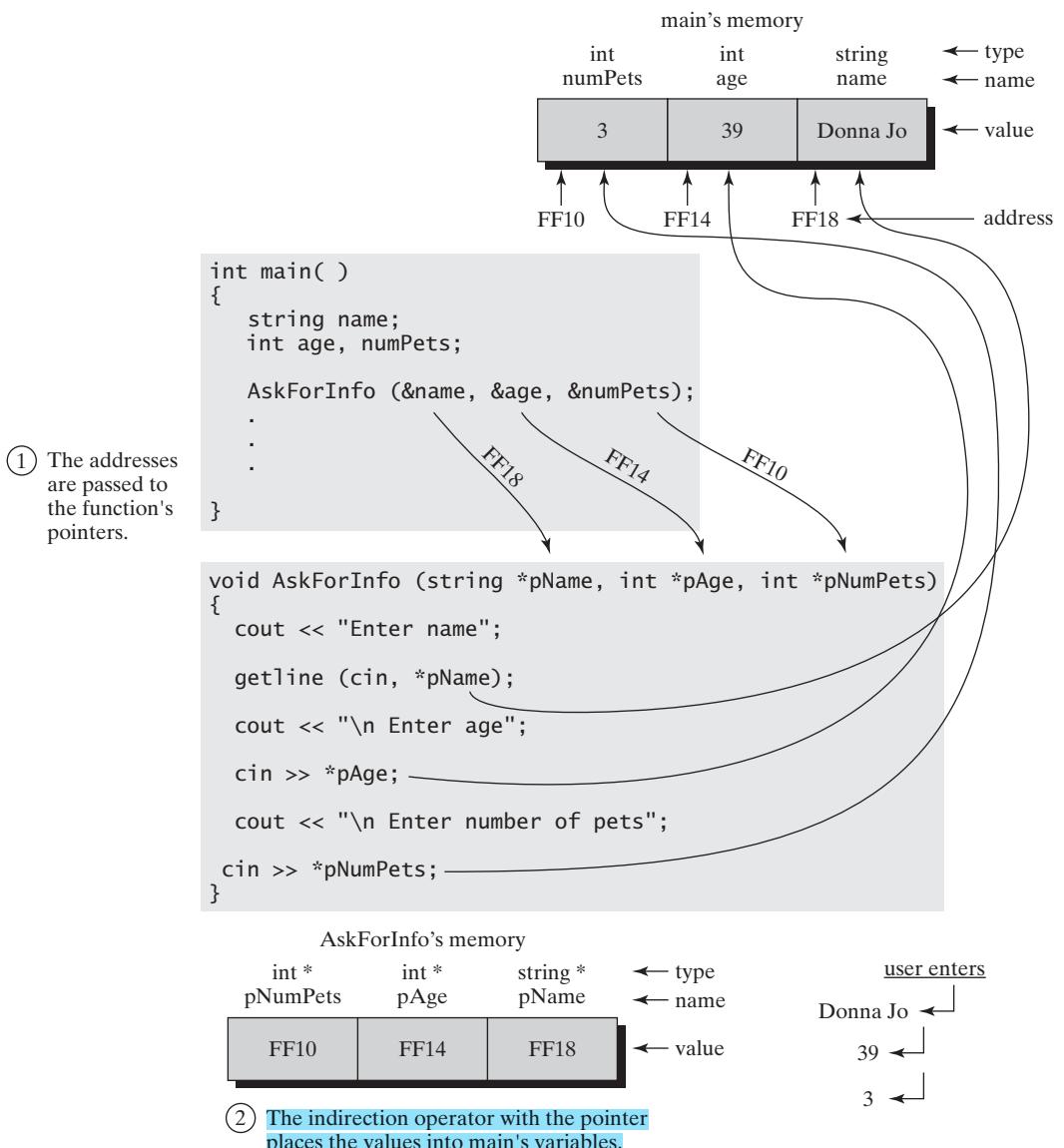
Let's look at a simple example to get us started with pointers. In Program 5-5 we have a function whose job is to ask the user for three data items: the user's name, age, and number of pets. We pass three addresses to the `Ask` function, which are stored in pointers in the function. See Figure 5-6 for clarification. In the function, we ask the user to enter the data, and we use the (*) indirection operator with the pointers. This action causes the data values to be written directly into the variables' locations in `main`. Hence we now have a technique that enables us to write functions that can “return” more than one data item.

function call-by-reference using pointers

term used when the address of a variable is passed into a pointer variable in the function

indirection operator, *

an operator, when used with a pointer, directs the program to the address held in the pointer variable

**Figure 5-6**

In a call-by-reference using pointers, addresses of main variables are explicitly passed to pointers. The indirection operator is used with the pointers to assign values into main's variables.

Program 5-5

```

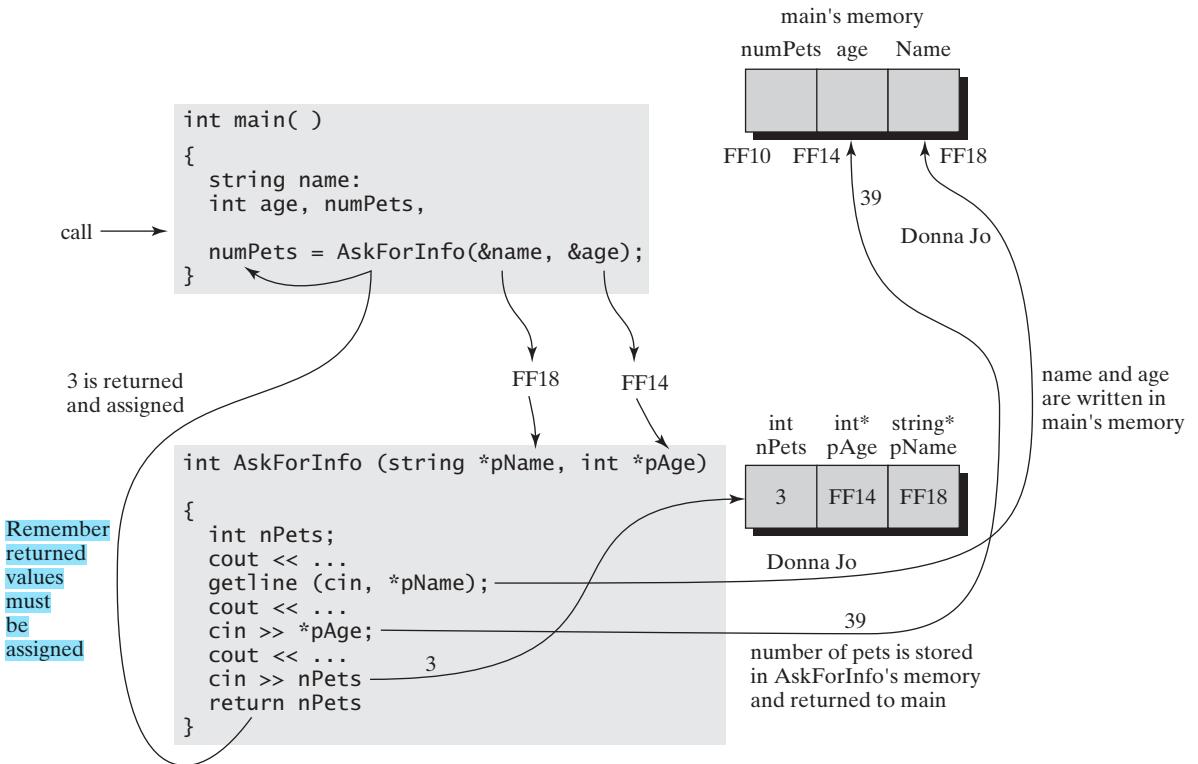
1 //A program with a function that uses
2 //pointers with indirection operators to
3 //return three pieces of information to main.
4
5 //include statements
6 #include <iostream>
7 #include <string>
```

```
8  using namespace std;
9
10 //function declarations
11 void AskForInfo(string *pName, int *pAge, int *pNumPets);
12 void WriteInfo(string name, int age, int numPets);
13
14 int main()
15 {
16     string name;
17     int age, numPets;
18
19     //Use the address operator to pass the variable
20     //addresses to the function.
21     AskForInfo(&name, &age, &numPets);
22
23     //No need to use pointers here. We can pass
24     //the values to the Write function.
25     WriteInfo(name, age, numPets);
26
27     return 0;
28 }
29
30
31 void AskForInfo(string *pName, int *pAge, int *pNumPets)
32 {
33     cout << "\n Please enter your name: ";
34     getline(cin, *pName);
35     cout << "\n Please enter your age: ";
36     cin >> *pAge;
37     cout << "\n Please enter the number of pets that you own: ";
38     cin >> *pNumPets;
39     cin.ignore();
40 }
41
42 void WriteInfo(string name, int age, int numPets)
43 {
44     cout << "\n Hi " << name << "!\n I see that you are "
45         << age << " years old and have " << numPets << " pets."
46         << endl;
47 }
```

**Output**

```
Please enter your name: Donna Jo
Please enter your age: 39
Please enter the number of pets that you own: 3

Hi Donna Jo!
I see that you are 39 years old and have 3 pets.
```

**Figure 5-7**

If a function returns a value to the calling function, you need to be sure to assign it in the call statement.

Before we leave this example, please notice that we do not use the keyword, `return`, in this function. There is no need because we are using three pointers. We could rewrite this program so that our `AskForInfo` function does return one value. For example, assume we are going to return the number of pets, and use pointers for the name and age. The program would need to be modified in this way. Refer to Figure 5-7. Whenever a function returns a value via the `return` statement, you need to be sure to assign the value to a variable. Here we assign the numbers of pets in the call statement.

```
//The prototype has a return value of int, and two address inputs.
int AskForInfo(string *pName, int *pAge);

//The call statement is written this way:
numPets = AskForInfo(&name, &age);

//The function is written like this:
int AskForInfo(string *pName, int *pAge)
{
    int numPets;
    cout << "\n Please enter your name: ";
    getline(cin, *pName);
```

```

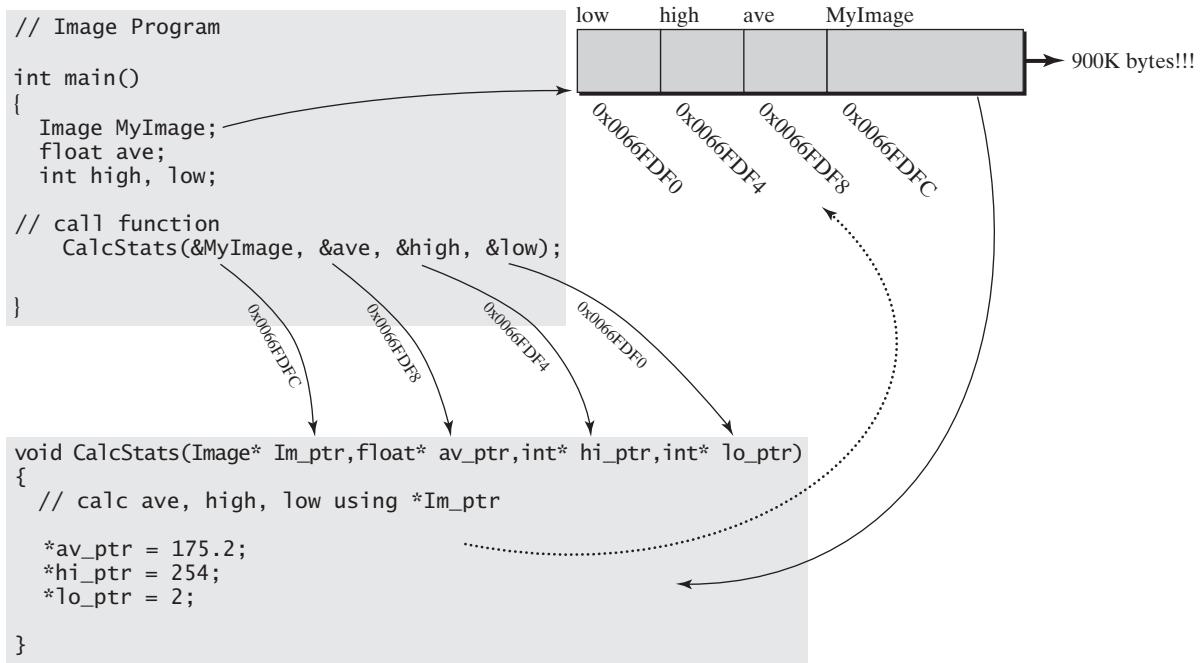
cout << "\n Please enter your age: ";
cin >> *pAge;
cout << "\n Please enter the number of pets that you own: ";
cin >> numPets;
return numPets;
}

```

Efficient Handling of Large Data Items

When a programmer works with large data items in a program, such as data records, image files or complicated scientific data, pointers can be quite useful in saving time and memory. Normally, if a variable is passed to a function, the function makes its own copy of the data. This practice is fine for a few parameters or a small amount of data; however, if the data item is large (i.e., a 640 by 480 pixel color image can easily be 900K bytes), it is best to have only one copy of the data in the program.

Figure 5-8 illustrates a program that contains a large digital image. (C++ allows us to create our own custom data types, so assume that we know how to create the data type *Image*.) If our program needs to calculate the statistics for the image—such as average pixel value, high, and low pixel—we can send the address



The main's Image data is accessed using the **Im_ptr*, and stats are calculated and written directly into main's variables.

Figure 5-8

In the Image Program, an address of an Image item is passed to the *CalcStats* function.

**call-by-reference
with reference pa-
rameters**

term used when the address of a variable is passed to a reference variable in the function

**reference pa-
rameter**

parameter declared using the & operator and acts as an implicit pointer

5.5

Functions and References

C++ provides programmers with a new call-by-reference technique for passing a variable's address to a function that uses reference parameters. A reference parameter is an address; i.e., it is implied though not plainly expressed as an address. In the **call-by-reference with reference parameter** passing technique, the function prototype and function header line contains **reference parameters**—that is, the & is used instead of the *. The & operator in the prototype and function header line is saying “Reference” (as opposed to “Pointer”). In the call statement, just the variable name is used; you do not need to use the & operator.

In this new type of call-by-reference, the addresses are passed to the called function and it accesses the calling function's data. However, the reference parameters make it possible for the call statement to have the call-by-value notation. In other words, a call-by-reference with reference parameters really is sending an address to a function, but the program does not need to use the indirection operator to access data.

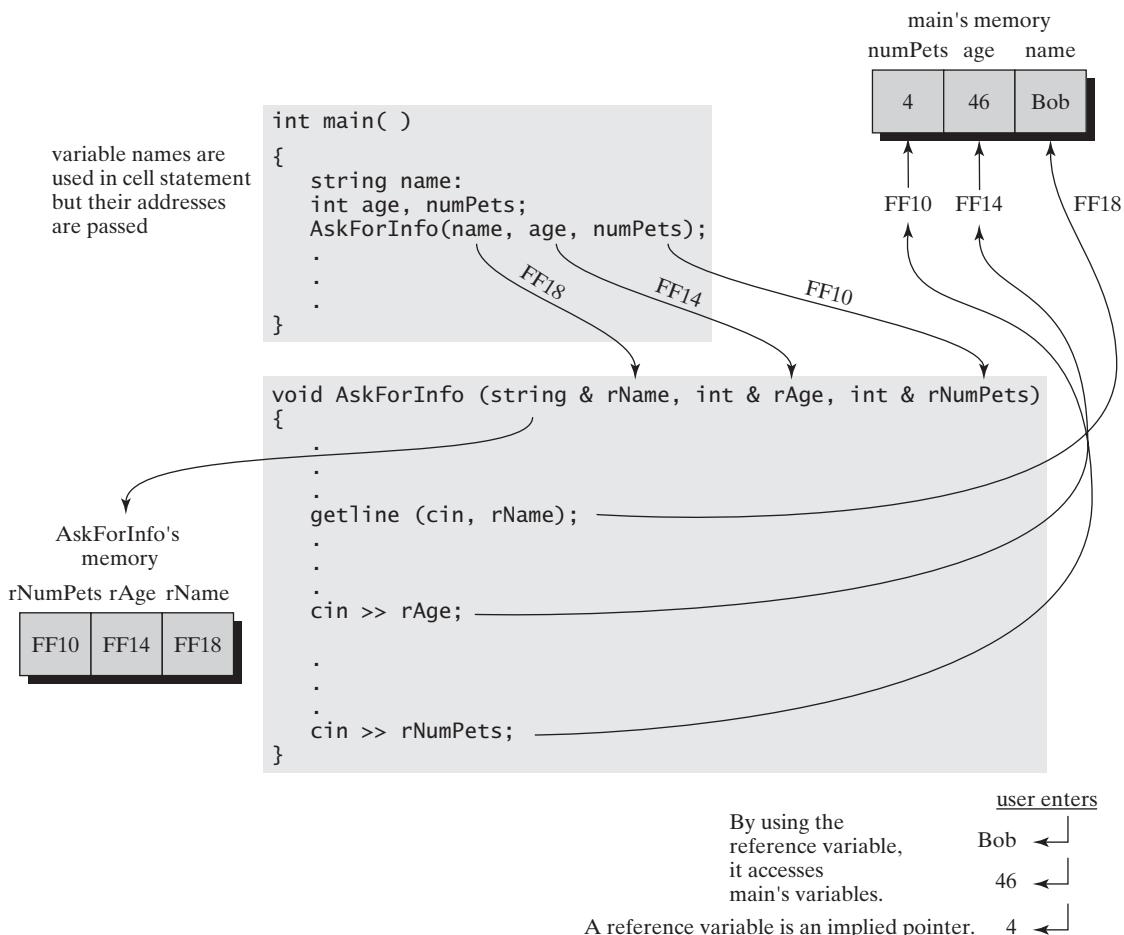
Reference parameters employ the reference operator, &, in the function declaration. Basically, the reference parameter is saying to the program, “This is really an address. Treat it like one. I'm not going to bother with the address or indirection operators because you and I both know this is an address.” Program 5-6 uses reference parameters in its *Ask* function. Notice how the & operator is used in the function prototype and function header line. The call statement has the variable names in it, but their addresses are being passed to the function. Look at Figure 5-9 for further clarification.

Program 5-6

```

1 //A program with a function that uses
2 //reference variables to return
3 //three data items to main.
4
5 //include statements
6 #include <iostream>
7 #include <string>
8 using namespace std;
9
10 //function declarations
11 void AskForInfo(string &rName, int &rAge, int &rNumPets);
12 void WriteInfo(string name, int age, int numPets);
13
14 int main()
15 {
16     string name;
17     int age, numPets;
```

of the *MyImage* to the *CalcStats* function. The *CalcStats* can then access the *MyImage* data and return the desired values.

**Figure 5-9**

In a call-by-reference using references, the variable names are placed in the call statement. Their addresses are passed to the function's reference parameters. When using the reference variable names, the function accesses main's variables directly.

```

18
19     //Use the variable names in the call to Ask.
20     //The addresses are passed since it is a
21     //call-by-reference.
22     AskForInfo(name, age, numPets);
23
24     //The Write function is a call-by-value.
25     //Copies of the values are passed to it.
26     WriteInfo(name, age, numPets);
27
28     return 0;
29 }
30

```

```
31 void AskForInfo(string &rName, int &rAge, int &rNumPets)
32 {
33     cout << "\n Please enter your name: ";
34     getline(cin, rName);
35     cout << "\n Please enter your age: ";
36     cin >> rAge;
37     cout << "\n Please enter the number of pets that you own: ";
38     cin >> rNumPets;
39     cin.ignore();
40 }
41
42 void WriteInfo(string name, int age, int numPets)
43 {
44     cout << "\n Hi " << name << "!\n I see that you are "
45         << age << " years old and have " << numPets << " pets."
46         << endl;
47 }
```

Output

```
Please enter your name: Bob
Please enter your age: 46
Please enter the number of pets that you own: 4
Hi Bob!
I see that you are 46 years old and have 4 pets.
```

Reference Parameter Limitations Incorporating reference parameters in calls to functions makes a programmer’s life easy—no address operators in the call statement, no indirection operators (*) in the called function. However, there are several limitations with references. You may not reference another reference. (It is possible to have a pointer to a pointer.) A reference variable must be initialized when declared if it is not part of a function parameter list, or if it is not a return value or class member. It is not possible to create a pointer to a reference or to create an array of references. Every tool has its optimum purpose, and reference parameters work well for functions.

Review: Let’s See Them Both Together

Let’s see an example that uses both pointers and references. This program is boring on the outside but exciting on the inside! There are two functions, each asks the user to enter an integer, double, and string value. One function uses pointers with the indirection operator, and the other uses reference parameters. In both cases, the functions receive the variable addresses. In the *AskUsingPtrs* function the addresses are explicitly passed in the call statement using the & operator. In the *AskUsingRefs* the address is implied. Examine the code in Program 5-7 (Here’s a question for you. Is it possible to write a function using both pointers and references?)

Program 5-7

```
1 //Demonstrate the both pointers and references
2 //in one program.
3
4 #include <iostream>
5 #include <string>
6
7 using namespace std;
8
9 void AskUsingPtrs(int *pInt, double *pD, string *pStr);
10 void AskUsingRefs(int &rInt, double &rD, string &rStr);
11
12 int main()
13 {
14     //declare 6 variables
15     int n, m;
16     double u, v;
17     string s1, s2;
18
19     //Pass the addresses to the pointers.
20     AskUsingPtrs(&m, &u, &s1);
21
22     //Use variable names when passing to references.
23     //The addresses are actually passed.
24     AskUsingRefs(n, v, s2);
25
26     cout << "\n In main Function";
27     cout << "\n The integers are: " << m << " and " << n ;
28     cout << "\n The doubles are: " << u << " and " << v ;
29     cout << "\n The first string is: " << s1 ;
30     cout << "\n The second string is: " << s2 << endl;
31
32     return 0;
33 }
34
35 //Call-by-reference using pointers and indirection operator.
36 void AskUsingPtrs(int *pInt, double *pD, string *pStr)
37 {
38     cout << "\n In AskUsingPtrs Function";
39     cout << "\n Please enter an integer: ";
40     cin >> *pInt;
41
42     cout << "\n Please enter a double: ";
43     cin >> *pD;
44     cin.ignore();
45
46     cout << "\n Please enter a string: ";
```

```

47     getline(cin,*pStr);
48 }
49
50 //Call-by-reference using reference parameters.
51 void AskUsingRefs(int &rInt, double &rD, string &rStr)
52 {
53     cout << "\n In AskUsingRefs Function";
54     cout << "\n Please enter an integer: ";
55     cin >> rInt;
56
57     cout << "\n Please enter a double: ";
58     cin >> rD;
59     cin.ignore();
60
61     cout << "\n Please enter a string: ";
62     getline(cin,rStr);
63 }
```

Output

In AskUsingPtrs Function

Please enter an integer: 42

Please enter a double: 5.2312

Please enter a string: I love using pointers.

In AskUsingRefs Function

Please enter an integer: 86

Please enter a double: 0.1234

Please enter a string: I really love using references.

In main Function

The integers are: 42 and 86

The doubles are: 5.2312 and 0.1234

The first string is: I love using pointers.

The second string is: I really love using references.

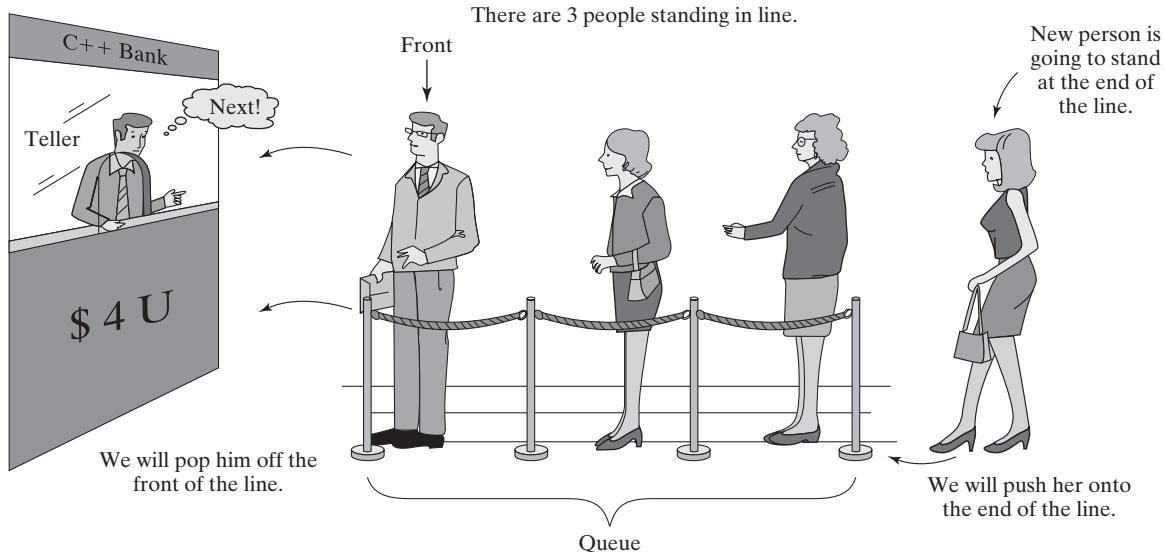
Why Pointers Are Important

As you become more proficient in writing functions and passing parameters between them, you may decide that using references is the better approach. For dealing with functions and obtaining more than one data value from a function, references are easier to use, and hence “better.” You may then ask yourself, “Why did we even learn about pointers if references are better?”

It is important for you to learn about pointers for two reasons. First, we introduce arrays in the next chapter. In order to understand fully the mechanism for passing arrays to functions, you need to understand what a pointer is. Often, in C++ reference documents, you’ll see the * in the data type description for arrays. We discuss this fully in Chapter 6. Secondly, pointers are required in order to have your program perform **dynamic memory allocation**. Dynamic memory allocation is the technique where your program reserves memory while it runs instead of

dynamic memory allocation

when the program reserves memory as the program executes

**Figure 5-10**

A queue is a class that models items that are “standing in line.” It is designed so that it follows the rules for lines.

reserving memory statically. What does this mean? Let’s say you need to read a file that contains many data values but you don’t know how many items are in the file. Instead of making a wild guess and declaring a large number of variables (that you may not need), your program merely allocates memory as it runs reserving memory for the actual number of values in the file. Dynamic memory allocation is an advanced topic, which you’ll be using in future C++ courses.

5.6

More Fun with C++ Classes: the queue Class

When was the last time you had to stand in a line? Maybe today you waited your turn at the gas station or the coffee shop. What are the rules you follow as you stand in a line? When you join a line, you are at the end of the line. The person at the front of the line gets to be served next, be it buying gasoline or ordering his coffee. The first person in the line is the first person to leave the line. This is known as a First In First Out (FIFO) format. Figure 5-10 illustrates a line at the C++ bank.

C++ provides a *queue class*, which models a FIFO line. The queue class provides several functions that we’ll use as we work with queue objects. Table 5-2 shows a partial list of these functions. As with a vector, you must declare a queue object and include the data type or class of items that are in the queue. The queue functions include a *push()* and a *pop()*. When you add someone to a queue, you “push” him into the line, and he is placed at the end of the line. When you remove the first person from the front of the line, that person is “popped” off the line. The push and pop concept and terminology are commonly used in computer science.

queue class

a C++ class that can be used to hold items, following the rules for standing in a line

TABLE 5-2

C++ queue class function

Function	Purpose
<i>size()</i>	Returns the number of items in the queue
<i>empty()</i>	Returns a true if the queue is empty.
<i>front()</i>	Returns the first item that is in the queue, but does not remove it from the queue.
<i>push()</i>	Pushes the item onto the end of the queue. When you push an item, it adds it to the queue.
<i>pop()</i>	Removes the first (front) item from the queue.

In Program 5-8, we model people standing in line at the C++ Bank. When we make the *bankLine* queue object, we tell it that it will be holding strings—the names of the people in line. Notice how we use reference parameters with functions that work with the queue. We pass the queue reference to the functions (instead of a copy of it). This technique has the functions accessing and modify main's *bankLine* queue directly.

Program 5-8

```

1 //This program models standing in line
2 //at the C++ bank. It uses a C++ queue object
3 //to hold the customers' names.
4
5 #include <iostream>
6 #include <queue>           //for queue object
7 #include <string>
8
9 using namespace std;
10
11 //function declarations
12 int ShowMenu();
13 void ShowLineInfo(queue<string> &rLine);
14 void AddToTheLine(queue<string> &rLine);
15 void ServeFirst(queue<string> &rLine);
16
17 int main()
18 {
19     cout << "\n Welcome to the C++ Bank ";
20     queue<string> bankLine;
21
22     int choice = 0;
23     while(choice != 4)
24     {
25         choice = ShowMenu();
26         switch(choice)

```

```
27     {
28         case 1:                      //show line info
29             ShowLineInfo(bankLine);
30             break;
31         case 2:                      //add person to line
32             AddToTheLine(bankLine);
33             break;
34
35         case 3:                      //serve first person
36             ServeFirst(bankLine);
37             break;
38     }
39 }
40
41 cout << "\n Thanks for using the C++ Bank! " << endl;
42
43 return 0;
44 }
45
46 void ShowLineInfo(queue<string> &rLine)
47 {
48     int num = rLine.size();
49     if(num == 0)
50         cout << "\n There is no one standing in line. " << endl;
51     else
52     {
53         cout << "\n Total customers in line: " << num << endl;
54         cout << rLine.front() << " is at the front of the line.";
55     }
56 }
57
58 void AddToTheLine(queue<string> &rLine)
59 {
60     string name;
61     cout << "\n Who do we add to the end of the line? ";
62     getline(cin, name);
63
64     //Use push() function to push the person onto
65     //the end of the queue.
66     rLine.push(name);
67 }
68
69 void ServeFirst(queue<string> &rLine)
70 {
71     //check if line is empty or not
72     if(rLine.empty() == true)
73         cout << "\n No one in line, let's have a coffee break.";
74     else
```

```

75      {
76          cout << "\n NEXT! Now serving " << rLine.front();
77
78          //Now pop this person off of the queue.
79          rLine.pop();
80      }
81  }
82
83  int ShowMenu()
84  {
85      int choice;
86      cout << "\n 1. Show Line Info. 2. Add Person"
87          << " 3. Serve person 4. Exit ==> ";
88      cin >> choice;
89      cin.ignore();           //strip off Enter key!
90      return choice;
91  }

```

Output placing two people in line, and serving them.

Welcome to the C++ Bank

```

1. Show Line Info. 2. Add Person 3. Serve person 4. Exit ==> 2
Who do we add to the end of the line? Mary Jane
1. Show Line Info. 2. Add Person 3. Serve person 4. Exit ==> 2
Who do we add to the end of the line? Hannah Banana
1. Show Line Info. 2. Add Person 3. Serve person 4. Exit ==> 1
Total customers in line: 2
Mary Jane is at the front of the line.
1. Show Line Info. 2. Add Person 3. Serve person 4. Exit ==> 3
NEXT! Now serving Mary Jane
1. Show Line Info. 2. Add Person 3. Serve person 4. Exit ==> 3
NEXT! Now serving Hannah Banana
1. Show Line Info. 2. Add Person 3. Serve person 4. Exit ==> 3
No one in line, let's have a coffee break.
1. Show Line Info. 2. Add Person 3. Serve person 4. Exit ==> 4
Thanks for using the C++ Bank!

```

5.7

Summary

All variables in a C++ program are stored in physical bytes of memory located in the RAM of the computer. When you declare and assign a variable in a program, such as

```

double MyDouble = 4.63;
int x;
x = 7;

```

there are five properties associated with it: data type, name, address, value, and number of bytes reserved for it. Table 5-3 summarizes these properties.

TABLE 5-3

Properties Associated with Each C++ Variable Note: for example double x = 4.63;

Item	What Is It?	Example
Data type	Establishes what kind of data the variable will contain	<i>double</i>
Number of bytes reserved	Quantity of memory used for this type	<i>8 for a double</i>
Name	How the program normally will reference this variable	<i>x</i>
Value	The data held in that variable	<i>4.63</i>
Address	The hexadecimal address of the first byte	<i>Unknown, but we can find it using the & operator</i>

This chapter introduces the pointer and reference variables. Important concepts are listed below:

- A pointer and a reference is designed to hold a hexadecimal address.
- A pointer is declared using the * operator, a reference is declared using an &.
- During a call-by-reference using pointers, a variable's address is assigned explicitly into the pointer by use of the address operator, &. During a call-by-reference using references, a variable's name is used in the call statement, but its address is passed to the function.
- A pointer, when paired with the indirection operator, accesses the data located at the address that it contains.

There are several important design, memory, and speed issues in which pointers play a very important role. Program 5-9 shows the *AskForXandY* function using call-by-references using pointers and using references and Table 5-4 presents the

TABLE 5-4

Comparison of References and Pointers in Functions

Item	Reference	Pointer
prototype	<code>void AskForXandY(int &rX, int &rY);</code>	<code>void AskForXandY(int *pX, int *pY);</code>
function body	<pre>void AskForXandY(int &rX, int &rY) { cout <<"enter x and y" cin >>rX >>rY; }</pre>	<pre>void AskForXandY(int *pX, int *pY) { cout <<"enter x and y"; cin >>*pX >>*pY; }</pre>
function call	<pre>int main() { int x, y; AskForXandY(x, y); }</pre>	<pre>int main() { int x, y; AskForXandY(&x, &y); }</pre>

pertinent lines of code comparing how references and pointers are used with functions. This same, short program is used to illustrate the common errors encountered while writing functions with call-by-reference.

Program 5-9

```
1 //Demonstrate the both pointers and references
2 //in one program.
3
4 #include <iostream>
5 using namespace std;
6
7 void AskForXandY(int *pX, int *pY);
8 void AskForXandY(int &rX, int &rY);
9
10 int main()
11 {
12     int x, y;
13     //Pass the addresses to the pointers.
14     AskForXandY(&x,&y);
15
16     //Use variable names when passing to references.
17     //The addresses are actually passed.
18     AskForXandY(x,y);
19
20     return 0;
21 }
22
23 //Call-by-reference using pointers and indirection operator.
24 void AskForXandY(int *pX, int *pY)
25 {
26     cout << "\n enter x and y  ";
27     cin >> *pX >> *pY;
28 }
29
30 //Call-by-reference using reference parameters.
31 void AskForXandY(int &rX, int &rY)
32 {
33     cout << "\n enter x and y  ";
34     cin >> rX >> rY;
35 }
36
```



An Overview of Common Errors Encountered with Pointers and References

Most of the errors illustrated here we have seen in previous chapters. Now that we have the `*` and `&` in our code, it makes the error messages all the more cryptic. Let's see the common ones here and realize it is just a new verse to an old song.

Compiler Error: Cannot Convert Parameter 1 from ‘int’ to ‘int *’

The “cannot convert” error seen here is due to forgetting to explicitly pass the address to a function with pointer inputs. In other words, the prototype is:

```
void AskForXandY(int *pX, int *pY);
```

The function is:

```
void AskForXandY(int *pX, int *pY)
{
    cout << "\n enter x and y  ";
    cin >> *pX >> *pY;
}
```

The call statement is written incorrectly as:

```
AskForXandY(x,y); // << WHOOPS! need the &x and &y
```

We are passing integer values in the call statement to a function whose input values are pointers! Hence the cannot convert error. When you see these type of errors, always check the prototype/function header line with the call statement.

Compiler Error: Cannot Convert Parameter 1 from ‘int *’ to ‘int &’

Another of the “cannot convert” error is due to accidentally putting the address operator into the call statement when you are passing by reference. The prototype is:

```
void AskForXandY(int &rX, int &rY);
```

The function is:

```
void AskForXandY(int &rX, int &rY)
{
    cout << "\n enter x and y  ";
    cin >> rX >> rY;
}
```

The call statement is written incorrectly as:

```
AskForXandY(&x,&y); // << WHOOPS! DON'T need the &x and &y
```

By placing the `&x`, `&y` in the call statement, we are passing integer pointers to the function. The function’s input values are references; therefore, the message indicating cannot convert `int*` to `int &` shows the confusion. Once again, check that your call statements match the input list!

Compiler Error: Illegal Indirection

This new error is not as dreadful as it first appears. Clicking on the error line shows that the complaining code is in the function’s `cin` statement. Here is the line of code in the pointer version of the `AskForXandY` function:

```
cin >> *pX >> *pY;
```

What could be the problem? We first check the prototype. We intend to use pointers. It looks good.

```
void AskForXandY(int *pX, int *pY);
```

The function call in main looks good too:

```
AskForXandY(&x, &y);
```

Ah, but upon careful inspection of the entire function, we see something is amiss. Can you spot the problem?

```
void AskForXandY(int pX, int pY)
{
    cout << "\n enter x and y ";
    cin >> *pX >> *pY;
}
```

The input list to this function shows two integers, NOT two integer pointers. (Examine the function header line.) If you forget to place the * in the function header line (sometimes due to copy/paste errors), you'll encounter the "illegal indirection" message.

5.8 Practice!

The programs in this section provide a mixture of reference parameters and pointers with functions. As you get more comfortable with the call-by-reference concept (either with reference parameter or pointers) writing your programs become easier, too.

Revisit *IsItPrime*?

In Program 5-10, we rewrite the *IsItPrime* program from Chapter 4, Program 4-12, page 216. Recall that in the earlier version, we ask the user to enter the whole number and we called a function to determine if the number was 1) positive, and 2) prime. A prime number is only divisible by 1 and itself. For example, 12 is not prime because it can be divided evenly by something other than 1 and 12 — namely 1, 2, 3, 4, and 6. But 7 is prime because it is only divisible by 1 and 7. In this new version of *IsItPrime* Part 2 our *IsItPrime* function not only returns a true or false value indicating prime status, but if it is not prime, it also tells us one divisor (other than 1 and itself.) The *IsItPrime* function has been designed to return a true/false value regarding the prime condition. We use a reference parameter to obtain the divisor value.

Program 5-10

```
1 //Is It Prime? Part 2
2 //Asks the user to enter a positive integer and
3 //determines if the number is prime.
```

```
4 //If it isn't prime, we obtain one of the numbers
5 //divisors, other than 1 and itself.
6
7
8 #include <iostream>
9 #include <string>
10 using namespace std;
11
12 //function declarations
13 int GetPositiveNumber();
14 bool CheckIt(int number);
15
16 //use reference variable to obtain divisor value
17 bool IsItPrime(int number, int &divisor);
18
19 int main()
20 {
21     int number, divisor;
22     string answer;
23
24     do
25     {
26         //get the number from the user
27         number = GetPositiveNumber();
28
29         //now check if it is prime
30         bool result = IsItPrime(number, divisor);
31
32         if(result == true)
33         {
34             cout << "\n\n The value " << number << " is prime. \n\n";
35         }
36         else
37         {
38             cout << "\n\n The value " << number << " is NOT prime.";
39             cout << "\n One divisor is " << divisor << endl;
40         }
41
42         cout << "\n Do another number? yes/no ";
43         getline(cin,answer);
44     }while(answer == "yes");
45
46     return 0;
47 }
48
49 int GetPositiveNumber()
50 {
```

```
51     int number;
52     bool OK;
53
54     do
55     {
56         cout << "\n Please enter a positive integer ==> ";
57         cin >> number;
58         cin.ignore(); //strip out the Enter key from queue
59         OK = CheckIt(number); //true if positive
60
61         if(OK == false)
62         {
63             cout << "\n Value is not positive, please re-enter ";
64         }
65     }while(OK == false);
66
67     return number;
68 }
69
70 bool CheckIt(int number)
71 {
72     if(number <= 0)      return false;
73     else return true;
74 }
75
76
77 bool IsItPrime(int number, int &divisor)
78 {
79     int remainder, ctr=2;
80     divisor = -1;           //set to -1 in case prime
81
82     //loop from 2 to n-1 and check remainder from modulus
83     //if a number doesn't have a remainder, there is a
84     //value that "goes into" it, therefore not prime
85     while(ctr < number )
86     {
87         remainder = number%ctr;
88         if(remainder == 0)
89         {
90             //ah ha! number has a divisor, not prime
91             divisor = ctr;
92             return false;
93         }
94         ctr++;
95     }
96
97     //since we divided the number by all values
98     //from 2 to n-1, and didn't have a 0 remainder
```

```
99      //the number is prime
100
101      return true;
102 }
```

Output

```
Please enter a positive integer ==> 12
The value 12 is NOT prime.
One divisor is 2
Do another number? yes/no yes

Please enter a positive integer ==> 7
The value 7 is prime.
Do another number? yes/no yes

Please enter a positive integer ==> 39
The value 39 is NOT prime.
One divisor is 3
Do another number? yes/no no
```

I Scream for Ice Cream Using Pointers

In Program 2-25, we calculated the volume of a single scoop ice cream cone. We assumed that the ice cream cone was a cone shaped and that the single scoop sat precisely in the top of the cone. In Program 5-11, we rewrite the ice cream calculation program using one function to obtain the three data items from the user: the cone diameter, height, and scoop flavor. We've used pointers here. Notice how we do not need pointers in the calculate function. We are passing two values to it, and it is returning one value back to main.

Program 5-11

```
1  //Ice cream volume calculation program
2  #include <iostream>          //for cout, cin, getline
3  #include <cmath>              //for power
4  #include <string>             //for string object
5
6  using namespace std;
7
8  //Practice using a #define to set PI's value
9  #define PI 3.14159265
10
11 void AskConeInfo(double *pDia, double *coneHt, string *pFlavor);
12 double CalculateIceCreamVol(double diameter, double coneHeight);
13
14 int main()
15 {
16     //Note: the ice cream cone and scoop have the same diameter.
17     double diameter, coneHeight;
```

```
18     double totalIceCream;
19
20     string flavor;
21
22     cout << "\n Welcome to the C++ Single Scoop Ice Cream Parlor";
23
24     AskConeInfo(&diameter, &coneHeight, &flavor);
25     totalIceCream = CalculateIceCreamVol(diameter, coneHeight);
26
27     //write output
28     cout.precision(1);
29     cout.setf(ios::fixed | ios::showpoint);
30
31     cout << "\n Results \n Your desired flavor is " << flavor;
32     cout << "\n Cone size " << diameter << " by " << coneHeight;
33     cout.precision(3);
34     cout << "\n Total volume of " << flavor <<
35         " ice cream is " << totalIceCream << " cubic inches " << endl;
36
37     return 0;
38 }
39
40
41 double CalculateIceCreamVol(double diameter, double coneHeight)
42 {
43     //Now calculate volume of ice cream
44     //First calculate the cone volume
45     double coneVol, radius;
46
47     radius = diameter/2.0;
48     coneVol = (PI * pow(radius,2) * coneHeight)/3.0;
49
50     //next calculate the volume of the scoop
51     double scoopVol = 4.0/3.0 * PI * pow(radius,3);
52
53     //total volume is entire cone plus 1/2 the scoop
54     double totalIceCream = coneVol + scoopVol/2.0;
55
56     return totalIceCream;
57 }
58
59 void AskConeInfo(double *pDia, double *pConeHt, string *pFlavor)
60 {
61     cout << "\n\n Please enter your desired flavor: ";
62     getline(cin,*pFlavor);
63
64     cout << "\n Please enter the cone height (inches): ";
65     cin >> *pConeHt;
```

```

66
67     cout << "\n Please enter the cone diameter (inches): ";
68     cin >> *pDia;
69     cin.ignore();
70 }
```

Output

Welcome to the C++ Single Scoop Ice Cream Parlor
 Please enter your desired flavor: chocolate mint
 Please enter the cone height (inches): 5
 Please enter the cone diameter (inches): 2.5

Results

Your desired flavor is chocolate mint
 Cone size 2.5 by 5.0
 Total volume of chocolate mint ice cream is 12.272 cubic inches.

Revisit Search for Woman's Name

Our Program 3-18 (page 156) used two vector objects to hold women's names and the names' origin. The program asked the user to enter a woman's name, and we searched the name vector for it. If we found the name, we displayed the origin of the name to the screen. We rewrite this program here using functions and references. We have a *FillVectors* and *SearchForName* functions that incorporate reference parameters. The *AskUserName* function asks for and returns a string object. To have fun with this program, it is written using three source code files, a NameFunctions.h and cpp, as well as a NameDriver.cpp. Look over the code here, and note how we have included the string and vector libraries in the NameFunctions.h file. We do this because the prototypes have vectors and strings. Remember to include whatever libraries are needed for each file—no more, no less.

Program 5-12

```

1 //NameFunctions.h
2
3 #include <string>           //string object
4 #include <vector>           //vector objects
5 using namespace std;
6
7 void FillVectors(vector<string> &names, vector<string> &origins);
8 string AskUserName();
9 bool SearchForName(vector<string> &names, vector<string> &origins,
10                   string &userName, string &nameOrigin );
```

Program 5-12

```

1 //NameFunctions.cpp
2 //Contains the function definitions for the
```

```
3 //prototypes contained in NameFunctions.h
4
5
6 #include <iostream>           //cout, cin, getline
7 #include <string>             //string object
8 #include <vector>              //vector objects
9 using namespace std;
10
11 bool SearchForName(vector<string> &names, vector<string> &origins,
12                     string &userName, string &nameOrigin )
13 {
14     //Search our name vector until we either find it, or
15     //run out of names. Stop looking when we find it.
16
17     int index = 0;
18     while(index < names.size() )
19     {
20         if(userName == names.at(index))
21         {
22             //once we find it, set nameOrigin and return to main
23             nameOrigin = origins.at(index);
24             return true; //found it.
25         }
26     else
27     {
28         ++ index;
29     }
30 }
31
32 //if we make it to this part of the function,
33 //we didn't find the name
34 return false;
35 }
36
37
38 void FillVectors(vector<string> &names, vector<string> &origins)
39 {
40     //Load our names and origins into the vectors
41     //These are main's vectors.
42     //We have reference variables for them.
43     names.push_back("Barbara");
44     origins.push_back("greek, meaning stranger");
45
46     names.push_back("Kelly");
47     origins.push_back("celtic, meaning warrior or defender");
48
49     names.push_back("Claire");
50     origins.push_back("french, meaning bright and clear");
```

```
51     names.push_back("Janis");
52     origins.push_back("english, God is gracious");
53
54     names.push_back("Ciara");
55     origins.push_back("celtic, meaning black and mysterious");
56
57     names.push_back("Lucy");
58     origins.push_back("latin, meaning bringer of light");
59 }
60
61
62
63 string AskUserForName()
64 {
65     string uName;
66     cout << "\n Please enter a woman's name, such as Kelly: ";
67     getline(cin,uName);
68     return uName;
69 }
```

Program 5-12

```
1 //Program asks the user to enter a woman's name.
2 //Search our vector to see if the name is in it.
3 //If it is, write out the origin/meaning of the name.
4
5 #include <iostream>      //cout, cin, getline
6 #include <string>        //string object
7 #include <vector>         //vector objects
8 using namespace std;
9
10 #include "NameFunctions.h"
11
12 int main()
13 {
14     cout << "\n Welcome to the C++ Name Search Program";
15
16     //Create 2 string vectors, names and origins.
17     vector<string> names;
18     vector<string> origins;
19
20     //Now use a function to fill the vectors with data
21     FillVectors(names, origins);
22
23     //Variables for name and results
24     string userName, nameOrigin, answer;
25     bool bFoundIt;
```



```

26
27      do
28      {
29          userName = AskUserName();
30
31          //the bFoundIt flag indicates if the name was found
32          bFoundIt = SearchForName(names,origins,userName,nameOrigin);
33          if(bFoundIt)
34          {
35              cout << "\n Here is your name: " << userName;
36              cout << "\n Origin: " << nameOrigin;
37          }
38          else
39          {
39              cout << "\n Sorry " << userName << " isn't in our vector.";
40          }
41
42          cout << "\n\n Do another name? yes/no ";
43          getline(cin,answer);
44
45
46      }while (answer == "yes");
47
48      return 0;
49  }

```

Output

```

Welcome to the C++ Name Search Program
Please enter a woman's name, such as Kelly: Ciara
Here is your name: Ciara
Origin: celtic, meaning black and mysterious
Do another name? yes/no yes

Please enter a woman's name, such as Kelly: Kristy
Sorry. Kristy isn't in our vector.
Do another name? yes/no no.

```

The C++ Song Organizer

One thing most programmers have in common is their love of fine music. Programmers like to be plugged into their players as they concentrate on their code. Not to be outdone, this next program provides a C++ Song Organizer that allows our user to enter the desired song list, and the songs are then “played back” in the given order. Of course, you are thinking that a queue is a perfect tool for this program!

In Program 5-13 we have a queue that contains string objects—which are song titles. The *FillPlayList* function asks the user to enter song titles. The *PlaySongs* function “plays” the tunes in the user’s order. Examine the code and output below to appreciate what a handy tool the queue class is for C++ programmers!

Program 5-13

```
1 //This program uses a queue to hold song titles that are
2 //input by the user. The PlayList function
3 //displays the front song and then removes it from the queue.
4
5
6 #include <iostream>
7 #include <queue>           //for queue object
8 #include <string>
9
10 using namespace std;
11
12 //function declarations
13 void FillPlayList(queue<string> &rPeaPod);
14 void PlaySongs(queue<string> &rPeaPod);
15
16 int main()
17 {
18     cout << "\n Welcome to the C++ Song Organizer ";
19     queue<string> PeaPod;
20
21     FillPlayList(PeaPod);
22     PlaySongs(PeaPod);
23
24     cout << "\n Thanks for listening! " << endl;
25
26     return 0;
27 }
28
29 void FillPlayList(queue<string> &rPeaPod)
30 {
31     cout << "\n Please enter desired song titles"
32         << " for your PeaPod. "
33         << "\n Enter \"done\" when finished." << endl;
34
35     string title = "not done";
36     while(title != "done")
37     {
38         cout << "Title: ";
39         getline(cin,title);
40
41         //push the title onto the queue
42         if(title != "done") rPeaPod.push(title);
43     }
44
45     cout << "\n You have entered " << rPeaPod.size()
46         << " titles for your listening pleasure. " << endl;
47 }
```



```

48
49 void PlaySongs(queue<string> &rPeaPod)
50 {
51     cout << "\n Now playing .... \n";
52
53     int totalSongs = rPeaPod.size();
54     for(int i = 0; i < totalSongs; ++i)
55     {
56         //play the song at the front of the line
57         cout << rPeaPod.front() << endl;
58
59         //now pop it out of the line
60         rPeaPod.pop();
61     }
62 }

```

Output

```

Welcome to the C++ Song Organizer
Please enter desired song titles for your PeaPod.
Enter "done" when finished.
Title: C You Later, Alligator
Title: Oh Say Can You C++
Title: Pop Goes the Programmer
Title: Programming in Moonlight Sonata in C# Minor
Title: done
You have entered 4 titles for your listening pleasure.

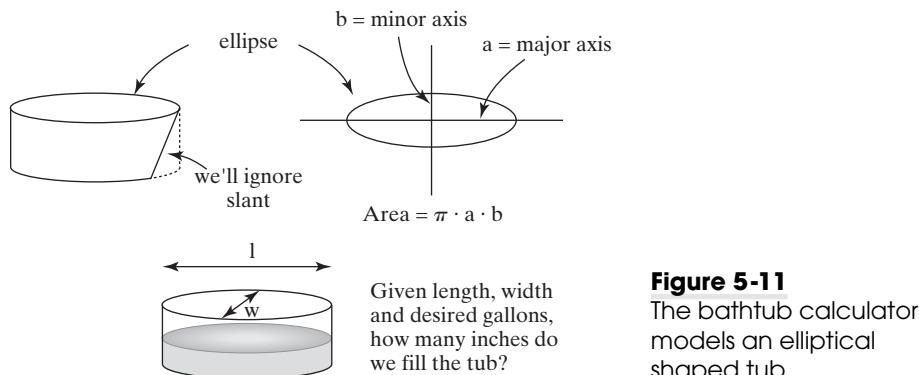
Now playing ....
C You Later, Alligator
Oh Say Can You C++
Pop Goes the Programmer
Programming in Moonlight Sonata in C# Minor
Thanks for listening!

```

Bathtub Volume Calculator

The southwestern United States has been experiencing drought conditions for several years. Some communities have imposed water use restrictions on the residents. In this last practice program we write a bathtub volume calculator that can be used to determine the water depth (in inches) for a bathtub, given a requested gallon use. In other words, if you know the dimensions of the bathtub, and you know that you only want to use so many gallons of water, this program will tell you the depth, in inches, in which to fill the tub.

Let's make this program interesting, and assume the bathtub has an elliptical shape. We'll ignore any slant on the edges. Figure 5-11 illustrates the geometry we'll use for this program. An ellipse has a major and minor axis (shown as a and b in the figure). The area of an ellipse is based on the circle area:

**Figure 5-11**

The bathtub calculator models an elliptical shaped tub.

$$\text{Area Ellipse} = \pi ab$$

For this program, we'll ask the user to enter the length and width of the elliptical tub and the number of gallons she desires to use for her bath. The bathtub calculator reports how many inches of water uses that number of gallons. The calculations are straightforward. Our approach is to determine the area in square inches of the bottom surface of the tub. Knowing this, we then determine the number of gallons is required for that 1 inch of water depth. The final step involves dividing total the number of gallons to be used by this one inch volume value. For example, let's say the bottom of the tub has an area of 1500 square inches. To fill the tub with one inch of water would require 6.49 gallons (231 cubic inches per gallon and $1500/231 = 6.49$). If our bather wanted to use 75 gallons for her bath, she could fill the tub to 11.55 inches ($75/6.49 = 11.55$). Examine the code in Program 5-14. It will help clarify solution.

Program 5-14

```

1 //This program determines the depth of water in inches
2 //for an elliptical bathtub, given the desired gallons.
3 //Area of an ellipse is PI*A*B where A/B are the major/minor axes.
4
5 #include <iostream>
6 using namespace std;
7
8 //prototypes
9 void AskEllipTubDim(float &rMajorAxis, float &rMinorAxis);
10 float AskTotalGallons();
11 float CalculateInches(float MajorAxis, float MinorAxis,
12                      float gallons);
13
14 int main()
15 {
16     cout << "\n Welcome to the C++ Elliptical Tub Water "
17         << " Use Program.";

```

```
18     cout << "\n We'll determine how many inches you"
19         << " can fill your tub "
20         << " \n given the desired max-gallon use.";
21
22     float majorAxis, minorAxis, gallons, inches;
23
24     AskEllipTubDim(majorAxis, minorAxis);
25     gallons = AskTotalGallons();
26     inches = CalculateInches(majorAxis, minorAxis, gallons);
27
28     cout << "\n Fill your tub " << inches << " inches to use "
29             << gallons << " gallons of water. " << endl;
30
31     return 0;
32 }
33
34 void AskEllipTubDim(float &rMajorAxis, float &rMinorAxis)
35 {
36     float length, width;
37
38     cout << "\n Enter elliptical tub dimensions:"
39             << "\n Tub length? (in inches) ";
40     cin >> length;
41
42     cout << "\n Tub width? (in inches) ";
43     cin >> width;
44     cin.ignore();
45
46     rMajorAxis = length/2.0;
47     rMinorAxis = width/2.0;
48 }
49
50 float AskTotalGallons()
51 {
52     float gallons;
53
54     cout << "\n Gallon-usage limit for this"
55             << " bathing experience? ";
56     cin >> gallons;
57     cin.ignore();
58
59     return gallons;
60 }
61
62 float CalculateInches(float MajorAxis, float MinorAxis,
63                         float gallons)
64 {
65     float PI = static_cast<float>(3.14159265);
```

```
66
67     //First calculate the area of the elliptical tub
68     float area = PI * MajorAxis * MinorAxis;
69
70     //Next determine how many gallons 1 inch contains
71     //231 cubic inches = 1 gallon of water
72     float gallonsPerInch = area/231.0;
73
74     float inches = gallons/gallonsPerInch;
75
76     return inches;
77 }
```

Output

Welcome to the C++ Elliptical Tub Water Use Program.
We'll determine how many inches you can fill your tub
given the desired max-gallon use.
Enter elliptical tub dimensions:
Tub length? (in inches) 66
Tub width? (in inches) 32
Gallon-usage limit for this bathing experience? 100
Fill your tub 13.9261 inches to use 100 gallons of water.

REVIEW QUESTIONS AND PROBLEMS

Short Answer

1. Name the five descriptive properties associated with each data variable in C++.
2. What does the address operator do?
3. Where are data variables stored when a program is running?
4. When a pointer variable has the indirection operator in front of it (like `*x_ptr`), what does this order tell the program?
5. Give three naming conventions for pointers. Describe another possible method for naming pointers.
6. Computer memory is addressed by using what type of notation?
7. In Visual C++ Express 2005, a double variable requires 8 bytes of storage space and an integer requires 4 bytes of storage space. How many bytes of memory are required for a double pointer? For an integer pointer?
8. Write a statement where a pointer and an indirection operator are used to assign a value to a variable. Write a statement where a data value is accessed by using the indirection operator–pointer combination.

9. For C++ programs, why is the stack an important part of the RAM?
10. Name two ways (show statements) in which a value can be accessed in a program.
11. When a variable's name is placed in the call statement, is it a call-by-value (i.e., is a copy of the variable's data being passed) or a call-by-reference (i.e., the address being passed)? Hint: this is a trick question.
12. The C++ queue class includes the *push()* and *pop()* functions. One works at the end of the line and one works at the beginning of the line. Explain which is which. Use the diagrams in Problems 21 and 22 to guide you as you create your diagram for this problem.
13. Explain three schemes that you could use to have a function "return" two data items to the calling function. Note: you can't use global variables.
14. When you have a *cin >>* statement that uses a pointer and indirection operator, or a reference parameter, where is the data being placed by the *cin* statement?
15. What is the input value to the *sizeof* operator? What does the *sizeof* operator return to the programmer?
16. Can a C++ program designate the exact location where the data variables are placed in memory? Explain.
17. How could a programmer make the variables in main available to other functions?
18. Is it possible to write a function that returns a pointer? What possible problem could the programmer encounter if the address of a function's local variable (i.e., pointer) was returned to the calling function?
19. Is it possible to write a function that contains a call-by-reference using pointers, call-by-reference using references, and a call-by-value? If you think it is possible, write the prototype, and sample call statement.
20. What is the benefit of using a call-by-reference when working with vectors or queue objects? In other words, why is it good to pass a reference of a queue or vector to a function instead of using a call-by-value?

Reading the Code

For Problems 21 and 22, show the hex addresses and variable values after the statements have been executed. (All pointers are 4 bytes!)

21. The first byte of memory below is xFF2A.

t	s	r	q	x	d	c	b	a

xFF2A

```

int main()
{
    float a= 32.5, b;
    int c = 5, d = 4.5, x;
    float *q, *r;
    int *s, *t;
    q = &a;
    r = &b;
    s = &x;
    t = &d;
    *r = c%d;
    *s = *t + c;
}

```

22. The first byte of memory is xFF30.

g_ptr	f_ptr	b_ptr	g	f	e	d	c	b	a

xFF30

```

int main()
{
    float a= 15.1, b, c;
    int d = 4, e = 18, f,g;
    float *b_ptr;
    int *f_ptr, *g_ptr;
    b_ptr = &b;
    f_ptr = &f;
    g_ptr = &g
    c = e%d + e/d;
    *f_ptr = 2.0*a - 1.0;
    *b_ptr = 7/9*(a*c + d);
}

```

Programming Problems

For all of the following programming problems, have the program write your name and program title to the screen one time. Incorporate a “play again” loop so that the user can repeat the program. Have the program ask if the user wishes to go again utilizing a string for the “yes” or “no” response. When the user is finished “playing,” have the program write out a goodbye message to the screen. Note: do not “trap” the user in a loop forcing him to enter “valid” data! For these problems, if you obtain invalid data, write an error and return to the “go again” code.

If you are comfortable building multi-file programs, do so.

23. Write a complete C++ program that has a *main* function, an *AskForTwoNumbers* function, and a *FindBigOne* function. The *main* calls the *AskForTwoNumbers* function, which asks the user for two integers. These two integers are “returned” to *main*, and are then sent to *FindBigOne*. It returns

the larger value to main. For example, if the user puts in 19 and 2, *FindBigOne* will return 19. If the two numbers are the same, it will send back either one.

24. Write a program that declares variables and pointers for a double, float, int, and short int (eight total variables). Assign values of your choosing to the numeric variables and the addresses of the variables into their pointers. Write out the addresses and values of each variable. Use the *sizeof* operator to state the number of bytes reserved for each type of variable. Then, using pencil and paper, diagram a memory using boxes that represent how the program allocated the memory.
25. Write a complete C++ program that declares four integer variables and four integer pointers. Assign the addresses into the pointers. Using the indirection operator with a pointer, assign the values 1, 2, 3, and 4 into the four integers. (Use the indirection operator with the pointers to write the integer values.) Your program should write the addresses and values of all eight variables to the screen. Use a tabular layout to list the variable's name, type, value, and address. Notice that the addresses of the integer variables are the values of the pointer variables.
26. Write a program that calls a function *AskRandHCy1*. This function asks for the radius and height of a cylinder. From *main*, pass these dimensions to the *CalcVolAndSA* function, which uses pointers or references to obtain the surface area and volume of the cylinder. Print the dimensions and calculated values from main showing four decimal places of accuracy. Refer to Chapter 2, Figure 2-2 and Figure 2-14. The volume of a cylinder is:

$$\text{Vol - Cylinder} = \pi r^2 h$$

where π is 3.14159265, r is the radius, and h is the height.

27. Can you stand to write another version of the Mortgage Calculation program? See Chapter 2, Problem 46 (page 74) and Chapter 4, Problem 28 (page 239). In this program, use either pointers or references where necessary. We'll first condense the data gathering process by writing the *AskLoanInfo* function that obtains the interest rate, principal, and number of years for the loan. You need to write two *MortCalc* functions (they should be overloaded, i.e., have the same name). The first *MortCalc* function is the one you wrote in Chapter 4. It is passed the three values and it returns a descriptive string that has all of the loan information. (The string should have the principal of the loan, interest rate, years, monthly payment, and total interest paid. Be sure dollar values are formatted to two decimal places and show the \$ sign.) The second *MortCalc* function is passed three loan parameter values as well as references or pointers so the function can “return” the monthly payment and total interest as separate, numeric values. In *main*, call these three functions and write the results to the screen. (It is true that you’re calculating the mortgage information twice but you’re gaining practice writing and using overloaded functions! Hint: Can you write a third *MortCalc* that is passed the three values and “returns” the monthly payment? Why is this a little tricky?)
28. We’re going to rewrite the C++ Fly A Kite Company delta kite production program. See Chapter 2, Problem 48 (page 76). In this program we’ll use a function named *AskForKiteInfo* that asks for the color scheme, bottom edge and center base-to-tip dimensions. From *main*, call the *Calculate* function,

which is passed the kite dimensions and it determines the surface area and side poles for the kite. Write all the kite information to the screen from *main*. Use either pointers or references in your functions where necessary.

29. Once again, we're going to go where no program has gone before, exploring new frontiers in vectors and references. We'll rewrite the planets program from Chapter 3, Problem 45 (page 173). From the user's viewpoint, this new version of the program is exactly like the Chapter 3 version. The big difference is how it is constructed. First thing to do is have the *main* function declare three planet vectors (name, distance from sun, and surface gravity). Using reference parameters, pass them to a *FillPlanetInfo* function that loads the vectors with the data from Table 3-8. Once loaded, these vectors are available to other functions (via pass by reference). Call the *AskSpaceTravelerInfo* function that politely asks our traveler to enter all the pertinent information including her Earth weight, travel speed, and the planet she wishes to visit. The *Calculate* function is passed pertinent data. It determines weight and travel time in hours. From calculate send all the data to a *Write* function that reports the data to the user. The *Write* function writes the user's name, destination planet, and the two weight values. The total hours is passed to the *WriteTime* function. This function converts the total hours to years, days and hours, then writes this data to the screen.
30. The vowel counting program in Chapter 3 (Program 3-20, page 160) asks the user to enter text into a string object and counts the number of vowels, punctuation marks, and spaces in the text. This program expands the vowel counting task by using a function designed to perform this counting task. First, have *main* obtain the string via an *AskUserForString* function. Main then passes it to the *Counter* function. This function performs the counts and reports them to the screen. To keep track of the totals there are seven total count variables which must be called-by-reference (either by references or by pointer variables). Instead of just counting the items in each string, this program needs to keep track of the total number of strings and totals for each vowel, punctuation and spaces as well. When the user has responded "no" to "go again?", report the totals.
31. This program models The C++ Modified Roulette game. The game works like this: the C++ Roulette wheel has the numbers 0–36 on it. You can place your bet in one of three ways:

bet on a number (pay off is 36 times the bet; \$1 gets you \$36),

bet on odd or even (pay off is 2 times the bet; \$1 gets you \$2), or

bet on a dozen—first 1–12, second 13–24, third 25–36 (pay off 3 times; \$1 gets you \$3).

The number zero does not count for odd or even or dozen, but can count as a number bet.

Your program must be designed using the following functions. (You may rename them if you like, but the functionality must be as described.) The *ShowInstructions* writes the game rules to the user, *ShowTheMoney* asks the user for the amount he'd like to bet and returns this money value. The *MakeABet* function asks the user what type of bet he'd like to place (number, odd/even or dozen). Depending on the response, you'll need to ask him relevant

questions. If the bet is a number, ask for which number. If the bet is a dozen, ask which dozen and if odd/even, which is it? The *SpinTheWheel* uses the random number generator, *rand()*, to simulate the wheel spin. It returns a number between and including 0–36. Last, the *FigureWinnings* is passed all the bet information, roulette number, and bet amount. It returns how much money the person has won. (Note: The person may win \$0.)

This program must use three boolean flags to keep track of which type of bet. (Your *MakeABet* function will set the correct flag to true, depending on the selected bet type.) You will also need separate variables for the bet number, which dozen and which odd or even. (You decide the type.) Report the wheel spin value, and the amount of money won.

32. Short-term parking at the C++ International Airport is based on a set fee of \$2.00 per 30 minutes. If you park for 15 minutes, the fee is \$2.00. If you park for 31 minutes, the fee is \$4.00. The lot opens at 4 AM and closes at midnight. This program will help our parking lot attendant calculate the parking fee for customers. There is an *AskTimeIn* function that asks the user to enter the time the vehicle entered the garage in the HR:MIN format. (Hint, use *cin* to obtain the values, *cin >> an int*, a char, and an *int*). There is an *AskTimeOut* function that asks the user to enter the time the vehicle left the lot. We'll make it easy and require the times to be based on the 24 hour clock meaning that 1 PM is 13:00, 11 PM is 23:00. The *ValidateTimes* function is passed the time in and time out values. It checks to be sure that the minute value is within 0–59 and the hour is within 0–23. It also checks that the time out is later than the time in. It has a boolean return value, indicating true for valid times, false for invalid times.

Your program should ask for the time in and time out, and validate them. When you have obtained valid time data, pass them to the *CalculateFee* function. It determines the parking fee along with the total time parked in hours and minutes. This data is returned to main and written to the screen. If the data is not valid, report this to the user. If the data is invalid, control should drop to the “do again?” question.

33. Let's rewrite the C++ Water and Ice program that calculates the bottled water that is delivered to warehouse grocery stores. See Chapter 2, Problem 53 (page 76). In this program we'll call a function named *AskForPallets*, which asks the store manager for the number of full and half-liter pallets desired. These two values are obtained by the *main* function and are passed to the *DetermineTotals* function. This function calculates the total number of cases for each drink, as well as total liters, gallons, and product weight. All five values are returned to *main*, where they are then written to the screen. Use either pointers or references in your functions where necessary.
34. The C++ Farm Irrigation Practice Program, 4-16 (page 231) is written using three separate *Ask* functions, which obtain the farmer's field and irrigation information. Rewrite this program so that it uses one *Ask* function to obtain this information. If you have already written Program 4-16, instead of rewriting all the questions, have your new *Ask* function call the three existing *Ask* functions. (No need to do unnecessary work!) Also, let's overload the *CalculateTimeAndGallons* function so that it makes the irrigation time and gallon values available to the programmer as numeric values. (You will need

to duplicate the calculation code in this function.) When you write the *main* function, call the single *Ask* function, then instead of obtaining the formatted string from 4–16’s program, call your new overloaded *Calculate* function. Write all the field data from *main*. Use either pointers or references in your functions where necessary.

35. The C++ Distillery problem in Chapter 3 (Problem 51, page 176) calculates the number of bottles needed for the whiskey that is aged in oak barrels. This program performs the same tasks, but the tasks are separated into these functions. Obtain the name of the whiskey using an *AskName* function. The *AskForBarrels* function obtains the diameter, height (in inches) and number of aging barrels. The *CalcNetWhiskeyVol* is passed the barrel information and uses the 5% loss due to the aging and filtering process to determine the total volume of whiskey in gallons. The volume is passed to the *DetermineBottles* function that calculates the number of two liter bottles and number of cases (twelve to a case). This function also determines the left over whiskey that the crew gets to sample after the bottling is finished. Call all of these functions from *main*. Use pointers or references where needed.
36. Recall from Chapter 4’s problem 40, (page 243) that our C++ programmer built a cabin on a hill. The cabin had a black, spherical tank to provide water for dishes, bathing, etc. The black tank provided solar heated water, but there was no temperature control. Now our programmer is remodeling her cabin and incorporating a two-cylindrical tank system, one white plastic (cold water) and one black plastic (hot water). Use a function to ask the user to enter the diameter and radius of the hot and cold water tanks. You may use one function, called twice, or one function that asks for two sets of information. (You can’t assume the tanks are the same size.) Determine the total capacity for the two tanks using a *Calculate* function. This calculate function determines a cylinder volume and should be called twice. Call *AskFlowRate* to obtain the rate of flow she experiences using this gravity fed system. (We’ll assume both tanks feed into a same sized lines and have a constant flow and is rated in gallons per hour.) Pass the total gallons and flow rate to the *Usage* function that determines the total number of hours and minutes of water available when both tanks are full. Report all data from *main*.
37. In Aesop’s Fables, “The Crow and the Pitcher,” a thirsty crow finds a pitcher of water that is half full. The crow puts his beak down into the pitcher, but cannot reach the water. The crow, smart bird that he is, drops pebbles into the pitcher, thus raising the water level high enough for him to quench his thirst. Let’s write this crow a C++ program so that he is able to determine how many pebbles he needs for this task.

First we need to know the size of the pitcher. We’ll assume a cylindrical shape because that’s easy to calculate. Have the crow enter the radius and height of the pitcher via the *AskPitcherDimension* function. Assume the crow has access to spherical marble-like pebbles of a uniform size. Obtain the diameter of the pebbles via the *AskPebble* function. We’ll need to ask for the crow’s name and the length of his beak (*AskCrowData*). Pass the necessary information to the *CalcPebbles* function. This determines how many pebbles the crow needs to drop into the pitcher so that he can submerge one-half inch of his beak into the water. Tell the crow the results by writing information from *main*.



Arrays

KEY TERMS AND CONCEPTS

array
array dimension
array index
array pointers
arrays and functions
atoi, *atof* functions
character string
C-string
element
filling arrays from data files
ifstream
multidimensional array
null character
null-terminated string
ofstream
out of bounds array
single-dimensional array
zero-indexed

KEYWORDS AND OPERATORS

array index ()

CHAPTER OBJECTIVES

Introduce the concept of an array—which is a list of variables with the same name and requires an index value.

Demonstrate how the C++ language automatically creates a pointer to any array that is declared.

Present single (one-dimensional) and multiply dimensioned (two-dimensional, three-dimensional, etc.) array concepts.

Illustrate how arrays are passed and used in functions.

Explore the problems of an array that is out of bounds in a C++ program.

Explain how to read data from a data file, a particularly useful tool when working with arrays.

Present example programs that read data from a file into an array and then pass the array to functions.



Run Faster! Jump Higher!

One part of writing software is designing the data variables to represent accurately the situation the program models. The software may be a computer game, an accounting program, an engineering data analysis package or drivers for a hardware device. Whatever the application, it is important to spend time during the program design phase thinking about what variables and their data types are needed.

In the previous chapters, we declared variables so that each one provided a location for storing one value. We had variables such as age, name, sum, and count. We asked the user his name and age, calculated PI and weekly pay values. In each case, we had one variable for each data value. Your work built up your confidence using different variable types. Now we raise the bar and say, "Run faster! Jump higher!"

In this chapter, we expand our C++ knowledge and learn how to declare a single variable that contains a list of data values of the same type. These new data types should be grouped logically i.e., they should relate naturally to each other. For example, you may declare a single variable that is actually a list of phone bills for one year. It's time to run faster, jump higher, and expand your C++ skills.

6.1

Using Single Data Variables

Let's start with a programming problem that shows how the ability to create a single variable containing a list of values can simplify life. Assume that you need to write a program that totals and averages the phone bills for a year, starting in January and ending with December. You need twelve variables—one for each month in the year—and you need to ask the user to enter these twelve values. Your program calculates and reports the average value. (Eventually it will be nice to have the program read the numbers from a data file instead of entering the data by hand. We will learn how to read data files later in this chapter.)

First, we must name our twelve variables. How about using abbreviations for the months as variable names to keep track of the twelve months? Then we ask for the numbers and calculate the average cost.

```

//Program 6-Incomplete program for finding the yearly total and
//average monthly phone bill.
#include <iostream>
using namespace std;
int main()
{
    // Declare 12 individual variables
    float jan, feb, mar, apr, may, jun;
    float jul, aug, sept, oct, nov, dec;
    float ave, yearSum;
    // Obtain monthly billing information
    cout << "\n Please enter your bill for January:";
    cin >> jan;
    cout << "\n Please enter your bill for February:";
    cin >> feb;
    cout << "\n Please enter your bill for March:";
    cin >> mar;
    // Program needs to ask for values for Apr to Dec here
    // This is left as an exercise for the reader. ;)
    // Now average
    sum = jan + feb + mar + apr + may + jun + jul
        + aug + sept + oct + nov + dec;
    ave = sum/12.0;
    cout << "\n Total yearly phone cost is $" << sum
    << "\n Average monthly phone bill is $"
    << ave << endl;
    return 0;
}

```

**array**

list of variables of the same data type referenced with a single name

array variable

a variable in C++ that contains a number of elements of the same type, referenced with same name

element

member of an array

array index

the integer value that references a specific element or member of an array

Writing a program in this manner is enough to drive anyone crazy. We will need twelve input statements. Then the average calculation takes three lines. Can you imagine what a hassle it would be to write out the twelve individual bills? There must be a better way to do this. Yes, there is. We will use an **array**.

6.2 Array Fundamentals

C++ allows the programmer to declare an **array variable** that groups together variables of the same data type, and references to this group of values can be made with a single name. Each array member or **element** is accessed via the array name and the **array index**. An array index is an integer value. The general format for an array declaration is:

```
dataType arrayName[ size ];
```

where *dataType* is *float*, *int*, *double*, etc., the *arrayName* is the variable name for the array, and *size* is an integer that represents how many variables are in this array. For the phone bills program, we declare an array of twelve floating point values:

```
float phone_bills[12];
```

The size is often referred to as the array dimension. When a programmer creates a list using one dimension (one size) value, as we did with `phone_bills`, it is referred to as a ***single-dimensional array*** and can be thought of as a single list or as a row of values. These values are stored contiguously in memory.

Figure 6-1 illustrates the `phone_bills` array. It is useful to visualize an array as a group of boxes. Each box represents a separate variable and each box has its own name. The array index is used to access the individual elements in the array. For example, when we assign the values into each element, we must use the array name and an integer index. This combination of array name plus index is how we refer to that specific variable location in memory.

It is possible to make arrays in C++ of any data type or class. For example, here are four more array declaration statements:

```
int numbers[1000];           // array of 1000 integers, named numbers
double rays[200];            // array of 200 double variables, named rays
string students[25];          // array of 25 strings, named students
char name[25];                // array of 25 characters
```

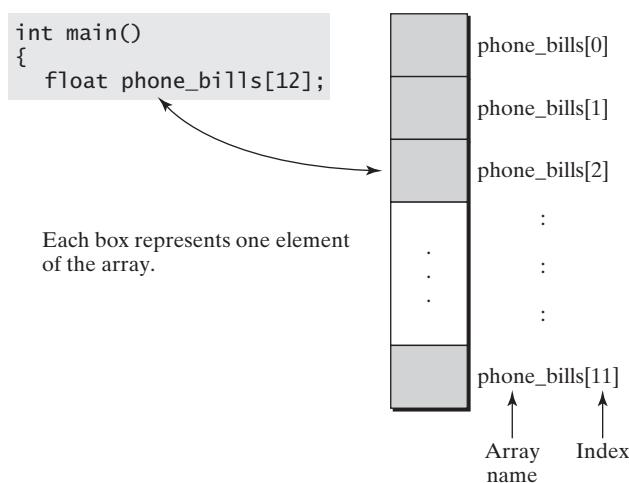
The last declaration (`char name[25];`) is an array of characters known as a ***null-terminated character string***. This name array is designed to hold up to 24 individual letters/characters with the 25th character being a null (a null is ASCII zero). These character strings or character arrays are also referred to as ***C-strings***, and were used in the C language to handle textual data in programs before the C++ string class was invented. (Remember that the C language was in use by the late 1970s and C++ didn't come into existence until the late 1990s.) The C-string was widely used, and is commonly found in C and C++ code today. We'll cover it later in detail in this chapter.

single-dimensional array

an array that represents a single list or column of values

null-terminated character string or C-string

a character array (C-string) that has a null character (zero) at the end of the pertinent data



Think of a one-dimensional array as a column of boxes — each box is referred to with name + integer index.

Figure 6-1

A single-dimensional array.

Arrays in C++ are Zero Indexed

zero-indexed:

when the first element of an array is referenced using a zero [0]

C++ arrays are referred to as *zero indexed*, which means that the array elements (boxes in Figure 6-1) are numbered starting with zero, *not* one! The name of the first element of any array has zero as the first index, and the last element's name is the size -1 index value. In the phone bill array, the first element (box) is *phone_bills[0]* and the last element is *phone_bills[11]*. Recall that a C++ string object's first character and a C++ vector's first element is indexed at 0, too.

Some programming languages, such as FORTRAN, allow the programmer to specify the starting array index. C++ does not allow this choice. Some beginning C++ programmers want to add an additional array element to the declaration and then ignore the first (index of zero) array element to make coding “easier.” This technique is not recommended. All arrays in C++ have the first index value of zero, and the last element is one less than the size. Do not create your own indexing scheme!

for Loops and Arrays and the Phone Bills Program

When writing software with arrays, the *for* loop is the programmer’s best friend. The *for* loop provides an efficient method for going through (traversing) an array. The index of the loop is used not only as a counter for the loop, it can also be used as the index value for the array. If you are not comfortable writing *for* loops, go back to Chapter 3, reread the *for* loop section, and look at the practice sample programs.

The Phone Bills program can be written with a *for* loop that makes coding much easier. The loop index is used to access each element of our phone bill array. Program 6-1 asks the user to enter the bill amounts for months 1 to 12. Figure 6-2 illustrates how the *for* loop index variable is used to access the array elements. Figure 6-3

```
// Obtain monthly billing information
for(i = 0; i < 12; ++i)
{
    cout << "\n Enter bill for month # " << i + 1 << "$";
    cin >> phone_bills[i];
}
```

The first time this loop runs, $i = 0$.
It asks the user for month number 1 ($i + 1 = 0 + 1 = 1$).
The user's value (i.e., 45.14) is placed in *phone_bills[0]*.

The second time this loop runs, $i = 1$.
It asks the user for month number 2.
The value the user enters (i.e., 45.14) is placed in *phone_bills[1]*.

The last time this loop runs, $i = 11$.
It asks the user for month number 12.
The value the user enters (i.e., 48.99) is placed in *phone_bills[11]*.

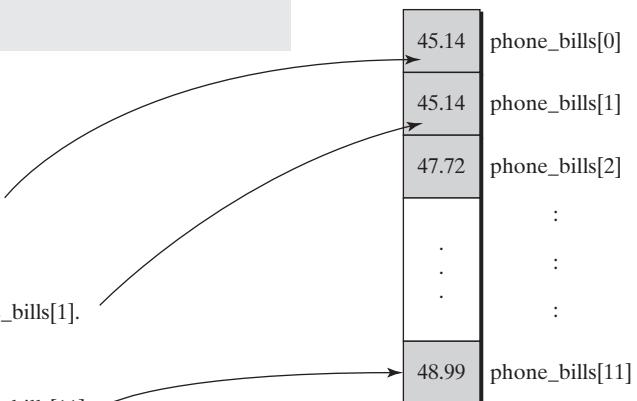


Figure 6-2

The *for* loop is a convenient tool for accessing array elements.

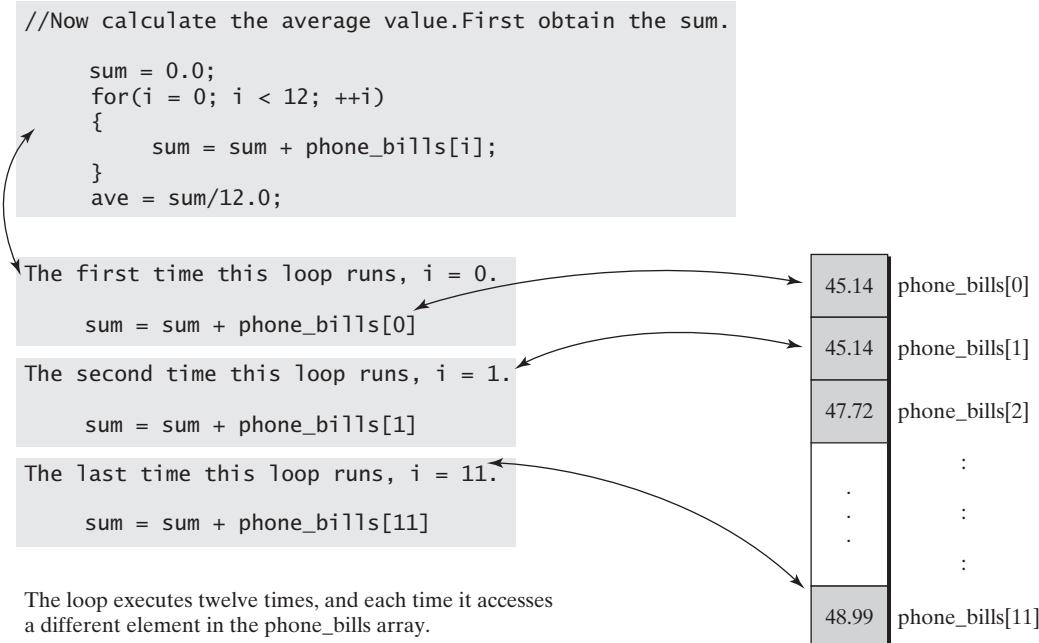


Figure 6-3
Average calculation of phone bills.

illustrates how this index is used again when calculating the yearly sum. The `for` loop index is a convenient tool to use whenever you need to go through an array.

Program 6-1

```
1 //A complete program for determining the average
2 //monthly phone bill over one year.
3
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     // Declare an array of 12 floats
10    float phone_bills[12];
11    float sum, ave;
12
13    int i;           //i is our loop index
14
15    cout << "\n A program that determines yearly total"
16    << "\n and average monthly phone bill.\n\n";
17
18    // Obtain monthly billing information
```



```

19      for(i = 0; i < 12; ++i)
20      {
21          cout << " Enter bill for month # " << i+1 << " $";
22          cin >> phone_bills[i];
23      }
24
25      // Now calculate the average value.
26      sum = 0.0;
27      for(i = 0; i < 12; ++i)
28      {
29          sum = sum + phone_bills[i];
30      }
31
32      ave = sum/12;
33      cout.precision(2);
34      cout.setf(ios::fixed);
35      cout << "\n Total yearly phone cost is $" << sum
36          << "\n Average monthly phone bill is $"
37          << ave << endl;
38
39      return 0;
40  }
```

Output

A program that determines yearly total
and average monthly phone bill.

```

Enter bill for month # 1 $45.14
Enter bill for month # 2 $45.14
Enter bill for month # 3 $47.72
Enter bill for month # 4 $48.43
Enter bill for month # 5 $45.14
Enter bill for month # 6 $45.14
Enter bill for month # 7 $51.78
Enter bill for month # 8 $49.73
Enter bill for month # 9 $45.14
Enter bill for month # 10 $52.65
Enter bill for month # 11 $45.14
Enter bill for month # 12 $48.99
```

```
Total yearly phone cost is $570.14
Average monthly phone bill is $ 47.51.
```

Array Declaration and Initialization

C++ does not automatically initialize array values to any value, just as it does not initialize singly declared variables. However, it is possible to initialize array values (i.e., assign values into the array elements) when the arrays are declared. The general form for initializing the values of a one-dimensional array at the time of declaration is:

```
dataType arrayName [size] = { list of array values };
```

The list of array values is a list of the initial values for the array elements. Commas must separate the values. The first value in the list is placed in the first element of the array (zeroth index), the next value is placed in the second element, etc. These values are not constant and can be changed by the program if needed.

Here is an example where an array of strings is declared and initialized in one statement:

```
string colors[5] = { "red", "blue", "white", "green", "yellow"};
```

The color “red” is placed in *colors[0]* and “yellow” is in *colors[4]*. We can assign numbers into an array of doubles. This array, named *compressionRatio*, has four values.

```
double compressionRatio[4] = { 3.21, 5.32, 0.87, 8.33};
```

A Nicer Version of the Phone Bills Program In Program 6-2, we write the Phone Bills program so that we use an array of C++ string objects to hold the names of the months. It is a nicer, more user-friendly version of the first program because it asks the user to enter the bill for the specific month. Notice how the month-names are initialized when the array is declared.

Program 6-2

```
1 //A program to calculate phone bill average for
2 //the year. It uses an array of C++ string objects
3 //to hold the month names.
4
5 #include <iostream>
6 #include <string>
7 using namespace std;
8
9 int main()
10 {
11     // Declare an array of 12 floats
12     float phone_bills[12];
13     float sum, ave;
14
15     //Declare and initialize the names of the months:
16     string month[12] = {"Jan", "Feb", "Mar", "Apr",
17                         "May", "June", "July", "Aug",
18                         "Sept", "Oct", "Nov", "Dec" };
19
20     int i;           //i is our loop index
21
22     cout << "\n A program that determines yearly total"
23         << "\n and average monthly phone bill.\n\n";
24
25     // Obtain monthly billing information
26     for(i = 0; i < 12; ++i)
27     {
```



```

28         cout << " Enter bill for month " << month[i] << " $";
29         cin >> phone_bills[i];
30     }
31
32     // Now calculate the average value.
33     sum = 0.0;
34     for(i = 0; i < 12; ++i)
35     {
36         sum = sum + phone_bills[i];
37     }
38
39     ave = sum/12;
40     cout.precision(2);
41     cout.setf(ios::fixed);
42     cout << "\n Total yearly phone cost is $" << sum
43         << "\n Average monthly phone bill is $"
44         << ave << endl;
45
46     return 0;
47 }
```

Output

A program that determines yearly total and average monthly phone bill.

```

Enter bill for month Jan $45.14
Enter bill for month Feb $45.14
Enter bill for month Mar $47.72
Enter bill for month Apr $48.43
Enter bill for month May $45.14
Enter bill for month June $45.14
Enter bill for month July $51.78
Enter bill for month Aug $49.73
Enter bill for month Sept $45.14
Enter bill for month Oct $52.65
Enter bill for month Nov $45.14
Enter bill for month Dec $48.99
Total yearly phone cost is $570.14
Average monthly phone bill is $ 47.51.
```

Array Out of Bounds == Big Trouble

C++ does not do any type of array boundary checking when a program uses arrays. The program does not warn you or stop the program if a statement causes the program to access an array element that is not legally declared. This ***out of bounds array*** feature of the C++ language can wreak havoc on your program when the program is executed.

In the simple case, accessing an out of bounds array element simply crashes your program or locks your computer, causing you to reboot your system. After the system has been restarted, you should be able to determine quickly where the

out of bounds array
attempting to access array elements that are not legally declared

problem occurs. (*Hint:* Use the debugger, step into your program, and look for an array operation with a *for* loop that is indexed incorrectly.)

In a not-so-simple (and therefore dreaded) case, the program performs the portion of the code where the actual illegal, out-of-bounds array access occurs. This action leaves the bad array intact, and the programmer believes that all is well. However, some innocent bystanders (data variables) are victims of this illegal operation. Usually the illegal array activity alters other variable value(s) because these variables are being written over accidentally. Such action will cause the software to run incorrectly. Often the programmer-investigator is led down the wrong path, examining the wrong variables.

To illustrate this bad array behavior, Program 6-3 sets up three integer arrays, all sized to 4 elements.

```
int High[4], Mid[4], Low[4];
```

We then use a *for* loop that is incorrectly written, causing the loop index value to be 0, 1, 2, 3, and 4.

```
for(i = 0; i <= 4; ++ i)
```

As this code executes, the three arrays are filled with random numbers within certain ranges. Our arrays each have four elements, but the *for* loop executes five times (*i*'s value ranges from 0 to 4). This *for* loop error causes us to access array elements outside the legal array boundary. Study Figure 6-4, which shows the memory configuration for these arrays and how C++ allows the program to write over other variable values.

The program crashes on line 24 when it attempts to access *High[4]*—which is reported as an access violation. (It goes outside the memory reserved for this program.) The reason we see any screen output is a timing issue in how fast the computer processes the *cout* statements versus reporting the access violation error. Try running this program yourself and playing with the sizes of the arrays. The more you see these types of errors, the more likely you'll be to avoid having them in your code.

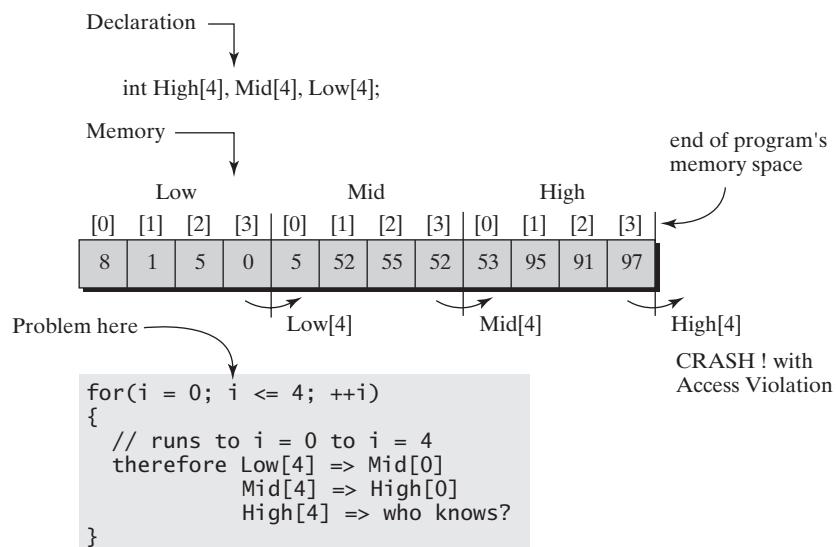


Figure 6-4

C++ does not check if you go out of bounds when accessing your array elements!

Program 6-3

```

1  //This program is bad.
2  //We go out of bounds on our arrays.
3
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9
10     //Make 3 arrays, all sized to 4
11     int High[4], Mid[4], Low[4];
12     int i;
13
14     //now fill using rand()
15     //make high values range between 90 - 100
16     //make low values range between 45-55
17     //make rain values range between 0 - 10
18
19     //YIKES, look at for loop index range   :-( 
20     for(i = 0; i <= 4; ++ i)
21     {
22         Low[i] = rand()%11;
23         Mid[i] = rand()%11 + 45;
24         High[i] = rand()%11 + 90;
25     }
26
27     cout << "\n Write the low values.";
28     for(i = 0; i < 4; ++ i)
29         cout << "\n Low[" << i << "] = " << Low[i];
30
31     cout << "\n Write the middle values.";
32     for(i = 0; i < 4; ++ i)
33         cout << "\n Mid[" << i << "] = " << Mid[i];
34
35     cout << "\n Write the highs values.";
36     for(i = 0; i < 4; ++ i)
37         cout << "\n High[" << i << "] = " << High[i];
38
39     return 0;
40 }
41

```

**Output**

Write the low values.

Low[0] = 8

Low[1] = 1

```
Low[2] = 5
Low[3] = 0
Write the middle values.
Mid[0] = 5
Mid[1] = 52
Mid[2] = 55
Mid[3] = 52
Write the highs values.
High[0] = 53
High[1] = 95
High[2] = 91
High[3] = 97 PROGRAM CRASHES HERE!
```

Comparing Vectors and Arrays

In Chapter 3 we learned how to use C++ vector objects, and mentioned (if you were familiar with arrays) that a vector is like a dynamic array. A C++ vector allows the programmer to add elements into a vector without limitation meaning that the vector can grow and shrink in size as the program executes. (“Dynamic” means changeable.) However, when you declare an array, it is fixed at that size. C++ does not allow you to make an array bigger as the program runs. The language sets up the requested memory and that is the “legal” memory in which you have to work.

You may ask the question, “Should I use arrays or vectors?” The answer depends on the program that you are writing. If you know exactly how many items you’ll have in a group, such as the yearly phone bill program, arrays work nicely. If your group grows as the program runs—that is, you’re not exactly sure how many items you’ll be working with—vectors are good to use. Now, just for fun, let’s rewrite the Phone Bills program so that it uses C++ vector objects instead of arrays. This allows you to compare the two techniques. Study Program 6-2 on page 299 and Program 6-4 here to see how we accomplish the same task using vector objects.

Program 6-4

```
1 //The Phone Bill Program using vectors to
2 //hold bill and month names.
3
4 #include <iostream>
5 #include <string>
6 #include <vector>
7 using namespace std;
8
9 int main()
10 {
11     // Declare vector of floats for phone bills
12     vector <float> vPhoneBills;
13
14     //Declare vector of strings for the months:
15     vector <string> vMonth;
```



```
16
17     //Must fill vector one at a time
18     vMonth.push_back("Jan");
19     vMonth.push_back("Feb");
20     vMonth.push_back("Mar");
21     vMonth.push_back("Apr");
22     vMonth.push_back("May");
23     vMonth.push_back("June");
24     vMonth.push_back("July");
25     vMonth.push_back("Aug");
26     vMonth.push_back("Sept");
27     vMonth.push_back("Oct");
28     vMonth.push_back("Nov");
29     vMonth.push_back("Dec");
30
31     float yearSum, ave;
32     int i;           //i is our loop index
33
34     cout << "\n A program that determines yearly total"
35     << "\n and average monthly phone bill.\n\n";
36
37     // Obtain monthly billing information
38     float bill;
39     for(i = 0; i < vMonth.size(); ++i)
40     {
41         cout << " Enter bill for month " << vMonth.at(i) << "$";
42         cin >> bill;
43         vPhoneBills.push_back(bill);
44     }
45
46     // Now calculate the average value.
47     yearSum = 0.0;
48     for(i = 0; i < vMonth.size(); ++i)
49     {
50         yearSum = yearSum + vPhoneBills.at(i);
51     }
52
53     ave = yearSum/12;
54     cout.precision(2);
55     cout.setf(ios::fixed);
56     cout << "\n Total yearly phone cost is $" << yearSum
57     << "\n Average monthly phone bill is $"
58     << ave << endl;
59
60     return 0;
61 }
```

6.3

Arrays and Functions

When C was a new language, computer memory was extremely limited and very expensive. The designers recognized that arrays could be very large, so a technique was implemented to handle arrays in an efficient manner. These C designers made it so that when an array is declared in a function, and memory allocated for it, this memory would (ideally) be the only copy of that data. Then, when the array “is passed to” a function, the language actually passes the address of it (i.e., a reference) to the function. The language wouldn’t make copies of the array when passing between functions. Let’s examine how the C designers accomplished this task.

A Free Pointer with Every Array

What do you suppose happens if we simply write out the name of the array? That is, we declare our array like this:

```
float phone_bills[12];
```

And we use the name in the `cout` statement, like this:

```
cout << "\n phone_bills = " << phone_bills;
```

Here is the output:

```
phone_bills = 0012FF44
```

What is going on here? This output looks like a hex address. Right! It is a hex address. The name of an array is actually a reference to the memory location of the array’s first element. (If you were thinking that the program would write out all the values in the array, that idea is incorrect. But you would have a lot of company with the same idea.) For numeric type arrays, if you use the array name in a `cout` statement, you’ll see the memory address. (*Special note:* A C-string/null-terminated character string is written to the screen just by using the array’s name. This is a special exception and we will soon see more on this. C-string/character arrays are handled differently than numeric arrays when doing input and output.)

Array Pointer

With the declaration of *any* array in C++, the language automatically generates an **array pointer**, which contains the memory location of the zero-element (i.e., the first element) of the array. For example (see Figure 6-5) when you declare the following:

```
float phone_bills[12];
```

your program does this automatically “behind the scenes.”

```
float *phone_bills = &phone_bills[0];
```

array pointer

pointer with the same name as the array that points to the first element of the array

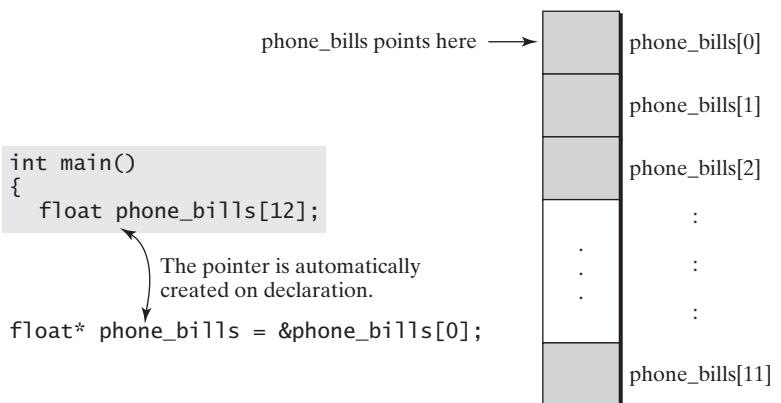


Figure 6-5

An array pointer is created automatically whenever an array is declared.

This pointer is available and is used when passing the array to a function. The usefulness of this pointer will become evident as we learn how arrays and functions work together.

Passing Arrays to Functions, The Original Call-by-Reference using References

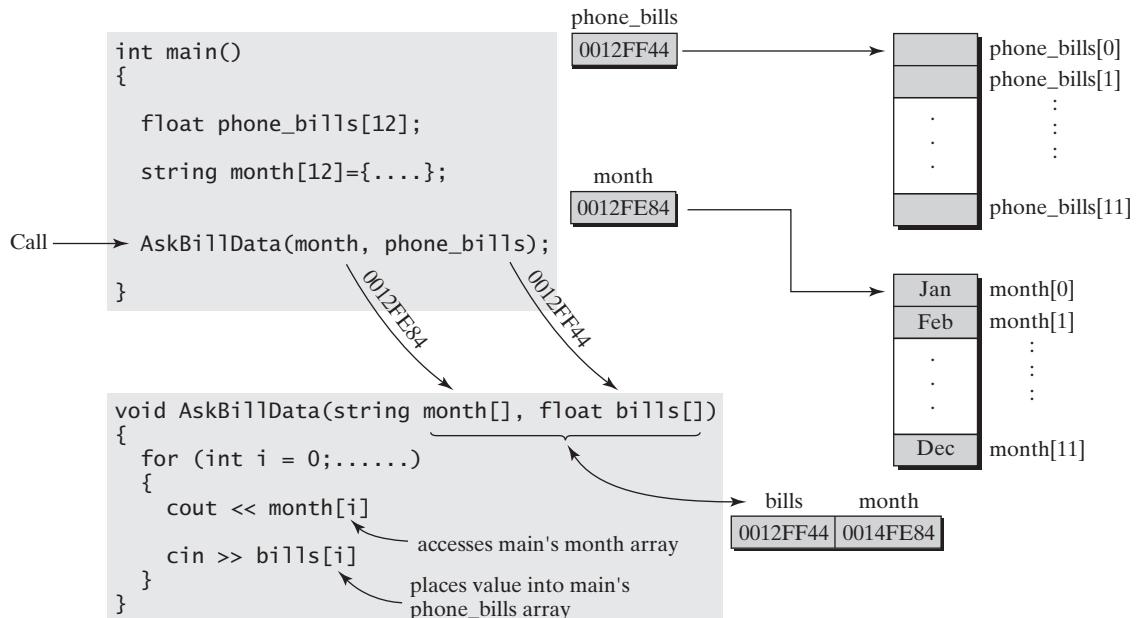
The name of the array is a reference (memory location) to the array's first element. When we need to pass an array to a function, we are actually passing this reference to it. We place the array name in the call statement, which passes the address to the function. (It is like a call-by-reference using references.). The program has only one copy of the array data; the function does not create its own local copy.

A function that uses an array must indicate the array data type in the prototype and function header line. Using the set of array operator brackets [] is how the program identifies the data type as an array. Once again, we use the Phone Bills program to illustrate passing arrays to functions. In Program 6-5 we use a function to fill the array, and one for the calculations (summing and averaging). Note how we use a reference to obtain the sum, and return the average value.

Figure 6-6 illustrates how the reference to the array is passed to the `AskBillData` function. Three important things to notice: 1) the function prototypes and function header lines indicate the array data type with `[]`, 2) the call statement has just the array name, and 3) because the array references in the functions are the functions' local variables and are merely holding addresses back to main's array, it is not necessary that the variable names match! Study Figure 6-6 and Program 6-5.

Program 6-5

```
1 //A program to calculate phone bill average for  
2 //the year. It uses two functions, one to fill the  
3 //array, one to sum and calculate the average.  
4  
5 #include <iostream>
```

**Figure 6-6**

The Phone Bills Program showing array addresses being passed to the `AskBillData` function.

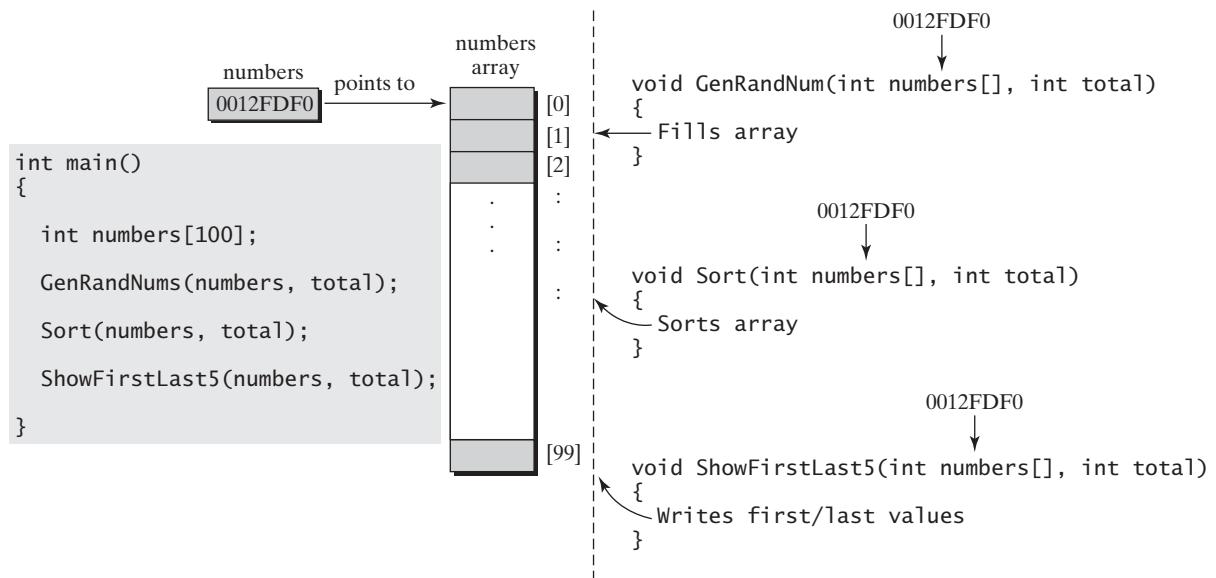
```

6  #include <iostream>
7  using namespace std;
8
9  //Function declarations use [] to indicate arrays
10 void AskBillData(string month[], float bills[]);
11
12 //return the average, use a reference for the sum
13 float CalcBillTotalandAve(float bills[], float &rSum);
14
15 int main()
16 {
17     // Declare an array of 12 floats
18     float phone_bills[12];
19
20     //Declare and initialize the names of the months:
21     string month[12] = {"Jan", "Feb", "Mar", "Apr",
22                         "May", "June", "July", "Aug",
23                         "Sept", "Oct", "Nov", "Dec" };
24
25     cout << "\n A program that determines yearly total"
26         << "\n and average monthly phone bill.\n\n";
27
28     //use array names in call statements
29     //(arrays addresses are actually passed to functions)
  
```

```
30     AskBillData(month,phone_bills);
31
32
33     float ave, sum;
34     ave = CalcBillTotalandAve(phone_bills, sum);
35
36     cout.precision(2);
37     cout.setf(ios::fixed);
38     cout << "\n Total yearly phone cost is $" << sum
39         << "\n Average monthly phone bill is $"
40         << ave << endl;
41
42     return 0;
43 }
44
45 //Function to ask for monthly bill data.
46 void AskBillData(string month[], float bills[])
47 {
48     // Obtain monthly billing information
49     for(int i = 0; i < 12; ++i)
50     {
51         cout << " Enter bill for month " << month[i] << " $";
52         cin >> bills[i];
53     }
54 }
55
56 //Function that determines bill average,
57 //given the yearly sum.
58 float CalcBillTotalandAve(float bills[], float &rSum)
59 {
60     rSum = 0.0;
61     for(int i = 0; i < 12; ++i)
62     {
63         rSum = rSum + bills[i];
64     }
65
66     return rSum/12.0;
67 }
```

Generating and Sorting Random Numbers Using Arrays and Functions

A common requirement when working with numeric arrays is to sort the values into order, either from low to high or high to low. Let's see another program where we generate 100 random numbers and store them in an array. We then use the Bubble Sort sort technique to arrange the array values into numeric order. Program 6-6, although simple, shows how the program has one array and the functions

**Figure 6-7**

Functions can fill, sort, and write array data.

all work on the data. Figure 6-7 illustrates how one function fills the array, one sorts it, and one writes the array values.

The Bubble Sort is a sort algorithm that is easy to understand, but painfully slow (in terms of computer speed rating). This sort works by comparing adjacent values in the array and begins with the first and second values. If the first value is larger than the second value, it swaps them. The sort then compares and swaps if necessary the next two values. This process continues through the entire array. The result of one pass has the largest value in the last element. The sorter then starts over and traverses the length of the array again placing the second highest value in the next-to-the-last element. There are many other, better sorts to use, but as a first look at sorting, this works for us.

Program 6-6

```

1 //A program that generates 100 random numbers
2 //in one function and uses a bubble sort to
3 //sort the array.
4
5 #include <iostream>
6 #include <ctime>           //for time function
7 #include <cstdlib>         //for rand()
8 using namespace std;
9
10 //functions need array [] in prototype
11 void Sort(int numbers[], int total);

```

```
13 void GenRandNums(int numbers[], int total);
14 void ShowFirstLast5(int numbers[], int total);
15
16 int main()
17 {
18     int numbers[100], total = 100;
19
20     cout << "\n Make 'em, Show 'em, Sort 'em, Show 'em again. ";
21
22
23     GenRandNums(numbers, total);
24
25     cout << "\n Here are the unsorted values.";
26     ShowFirstLast5(numbers, total);
27
28     Sort(numbers, total);
29
30     cout << "\n Here are the sorted values.";
31     ShowFirstLast5(numbers, total);
32
33     return 0;
34 }
35
36 //Generate a "total" number of random values.
37 void GenRandNums(int numbers[], int total)
38 {
39
40     //Seed the generator with the system time.
41     srand(time (NULL) );
42     for (int i = 0; i < total; ++i)
43     {
44         //rand() gives us values between 0 and 32767
45         numbers[i] = rand();
46     }
47 }
48
49 //Sort using the Bubble sort technique, slow but simple.
50 //Sorts array from low to high
51 void Sort(int numbers[], int total)
52 {
53     int i,j, temp;
54
55     //compare adjacent values, switch if out of order
56     for(i = 0; i < total-1; ++i)
57     {
58         for(j = 1; j < total; ++ j)
59         {
```



```
60         if(numbers[j-1] > numbers[j])
61         {
62             temp = numbers[j];
63             numbers[j] =numbers[j-1];
64             numbers[j-1] = temp;
65         }
66     }
67 }
68 }
69
70
71 void ShowFirstLast5(int numbers[], int total)
72 {
73     cout << "\n First 5 values \n";
74
75     for(int i = 0; i < 5; ++i)
76         cout << "numbers[" << i << "] " << numbers[i] << endl;
77
78     cout << "\n Last 5 values \n";
79     for( i = total - 5; i < total; ++i)
80         cout << "numbers[" << i << "] " << numbers[i] << endl;
81
82 }
```

Output

Make 'em, Show 'em, Sort 'em, Show 'em again.

Here are the unsorted values.

First 5 values

```
numbers[0] 18205
numbers[1] 32434
numbers[2] 12711
numbers[3] 2718
numbers[4] 28005
```

Last 5 values

```
numbers[95] 27244
numbers[96] 31614
numbers[97] 22379
numbers[98] 5097
numbers[99] 13456
```

Here are the sorted values.

First 5 values

```
numbers[0] 553
numbers[1] 572
numbers[2] 573
numbers[3] 608
numbers[4] 1254
```

```
Last 5 values
numbers[95] 31140
numbers[96] 31614
numbers[97] 32322
numbers[98] 32409
numbers[99] 32434
```

6.4

C-strings, also Known as Character Arrays

Most programs of any size contain textual data. This data can range from simple name and address information, file names, text input from the user, to complicated encryption schemes. If you are writing a program for automobiles, you may have text data in the form of make, model, color, vehicle identification number, and license plate number. We have learned how to use the C++ string objects to handle program text data. Now it is important to cover the high points of using C-strings, as they are still present in C/C++ code today.

The C-string is a one-dimensional character array, which is null-terminated. That is, the character array contains the characters and has an ASCII zero value, known as the **null character** ('`\0`', read as "backslash zero"), at the end of the character data. This null character indicates the end of the text data. It is important that you always remember to make your character arrays big enough to include the null character.

The basic format for declaring a null-terminated character string is

```
char variableName[ size ];
```

where the *variableName* is the name of the C-string, and size is an integer value that represents the number of characters in the string. Here we make character arrays for holding mailing address information.

```
char Name[50];
char Address[70];
char City[30];
char State[5];
char Zip[15];
```

Another way to declare this data is like this:

```
char Name[50], Address[70], City[30], State[5], Zip[15];
```

Notice that we make the zip code array 15 characters. Students begin to work with C-strings and they often make the mistake of trying to size their arrays exactly—such as declaring the Zip array to [5] or [10], or the *State*[2]. The problem is that you need to remember C++ needs the null character in the character arrays. Don't be cheap with your C-string array declarations! The char data type requires one byte per element! Always give yourself a bit of extra space, but don't go overboard.

null character:

"zero" character,
'`\0`'

Character String Initialization

Initializing character strings follows the same format as initializing numeric valued arrays. Although it is possible to initialize this string by using the comma-separated list, C++ allows you to use a string constant. Here we initialize the C-string to contain “I love C++!” When you declare and initialize a character array in this manner, the null character is automatically added for you. See Figure 6-8.

```
char line[15] = "I love C++!";
```

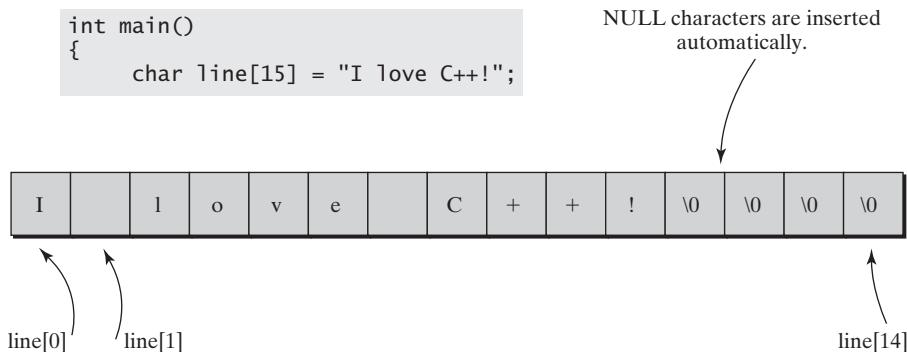


Figure 6-8
Initializing a C-string, nulls are added to the unused elements.

The Null Character

The C++ programmer must ensure that the character string has been null-terminated, which means that the character string has a null character (‘\0’) at the end of the data. The null character is a zero (‘\0’). C++ uses the null character to indicate where the end of the data is located.

For most of your day-to-day work with C-strings, the null character is placed into the array automatically. You don’t have to worry about it—however, it is important to remember that all of the C-string functions provided in the string library and the output functions in iostream require a null character at the end of the relevant data in order for your program to work correctly. If you are constructing a C-string from scratch, or forget to place data into it, you will end up with problems!

The following program shows how important the null character is when writing character data to the screen (or to a data file). In this example, we declare one C++ string, and three C-strings. We initialize the string object and one of the C-strings when declared. We place data into one of the C-string arrays, a letter at a time, and forget the null character. We do not place any data into the last C-string array. First, look at the code, and examine the output.

Program 6-7

```
1 //This program shows how to initialize
2 //C-strings and C++ string objects.
3 //It also shows the importance of the NULL char.
4
5 #include <iostream>
6 #include <string>
```



```
7 using namespace std;
8
9 int main()
10 {
11     //First use a C++ string
12     string strObject = "This text is in a string object!";
13
14     //Declare C-string and initialize.
15     char cppLove[20] = "I love C++!";
16
17     //Declare and fill individual elements.
18     //((whoops!) no null placed here
19     char noNullText[20];
20
21     noNullText[0] = 'W';
22     noNullText[1] = 'h';
23     noNullText[2] = 'o';
24     noNullText[3] = 'o';
25     noNullText[4] = 'p';
26     noNullText[5] = 's';
27     noNullText[6] = ' ';
28     noNullText[7] = 'n';
29     noNullText[8] = 'o';
30     noNullText[9] = ' ';
31     noNullText[10] = 'n';
32     noNullText[11] = 'u';
33     noNullText[12] = 'l';
34     noNullText[13] = 'l';
35
36     //Declare, but don't put anything in it
37     char noText[20];
38
39     //Now write all four:
40     cout << "\n Write out the four pieces of text data." << endl;
41     cout << "          String object: " << strObject << endl;
42     cout << "          C++ love: " << cppLove << endl;
43     cout << "          noNullText: " << noNullText << endl;
44     cout << "          No text: " << noText << endl;
45
46     return 0;
47 }
```

Output

Write out the four pieces of text data.

String object: This text is in a string object!

C++ love: I love C++!

noNullText: Whoops no null|||I|||I|||I|||I||| I love C++!

No text: Whoops no null I love C++!

Declarations

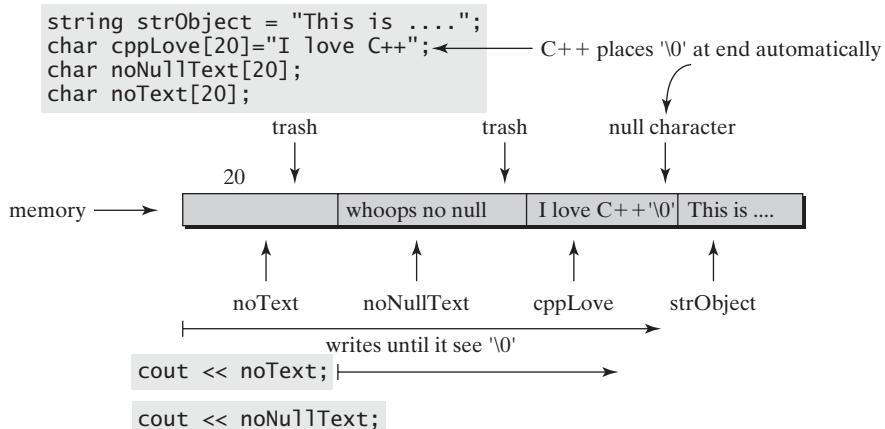


Figure 6-9
The null character indicates the end of the data in a C-string.

What is going on with this code? The `cout` object relies on seeing the null character when it writes C-string data to the screen—the null character says, “This is the end of the data.” Figure 6-9 shows how the four data items are stored in memory. The string object is at the far end of memory, the other three character arrays are stacked in so that the last declared array (`noText`) is at the near end of the memory. When we try to write out `noText` array, `cout` writes characters until it sees a null character. Thus we see 20 “trash” ||||| characters, then “Whoops no null,” then 6 “trash” characters, then “I love C++.” There is a null character ‘\0’ at the end of the `cppLove` character array—which `cout` uses as the stopping point. The same thing happens with the `noNullText` array. We see “Whoops,” trash, and the love text.

Whenever you see these “trash” characters ||||| in your output (either screen or file) you are writing character array data that does not have the null character, ‘\0’ at the end of the data. How would you fix the above program so that you don’t have this problem? Here’s a two-line hint:

```

noNullText[14] = '\0'; //at line 35
char noText[20] = ""; //initialize like this on line 19
  
```

C-string Input

The technique for reading data into character arrays is almost identical to those for reading data into C++ string objects. The rules for `cin` and `getline` hold true when reading C-string data. (The `cin >>` reads until the first space or enter key, and leaves the enter key in the queue. The `getline` reads until it sees an enter key and removes it from the queue.) The format for `getline` is a bit different, which we’ll examine here.

The How Old Are You, Claire? Program contains both character arrays and numeric variables. We’ll read the character data using a new form of `getline`. (The `getline` has several forms.) When reading from the keyboard, the prototype of `getline` that we need for C-strings is:

```
cin.getline(char* text ,int max);
```

The first parameter is the name of the character array. (Remember, array names in C++ are actually memory addresses—pointers—to the array.) The second parameter should be the size of the character array that is being read. The *getline* function reads until it sees an Enter key, or until it has read (max-1) *chars* into the array. It then places a null character in the last element. To read a name, use this form of *getline*. The code looks like this:

```
char name[50];
cout << "\n Enter your full name. ";
cin.getline(name, 50);
```

Program 6-8 reads data using *getline* and *cin*. The *cin.ignore()* function is used to strip off the Enter key left by *cin*.

Program 6-8

```
1 //Demonstrate how to read C-string character data and
2 //numeric data in a program.
3
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9
10    cout << "\n The How Old Are You Claire? Program "
11        << "\n asks our friend, Claire some personal questions.";
12
13    char husband[25], footballTeam[25];
14    int age, numKids;
15
16    //ask Claire for husband's name, her age,
17    //favorite football team and number of kids
18
19    cout << "\n Hi Claire, what is your husband's name? ";
20
21    //use getline to read into the character array
22    cin.getline(husband, 25);
23
24    //Now ask age
25    cout << "\n How old are you? ";
26    cin >> age;
27    cin.ignore(); //strip off Enter key left by cin
28
29    //Team name? Use another getline
30    cout << "\n What is your favorite football team? ";
31    cin.getline(footballTeam,25);
32
33    //Number of kids
34    cout << "\n How many kids do you have? "
```



```
35     cin >> numKids;
36     cin.ignore();
37
38
39     // write info out to screen
40     cout << "\n Hi Claire! Your husband is " << husband
41             << ".\n Your team is " << footballTeam <<
42             ". \n You are " << age << " years old and have "
43             << numKids << " kids." << endl;
44
45     return 0;
46 }
```

Output

```
The How Old Are You Claire? Program
asks our friend, Claire some personal questions.
Hi Claire, what is your husband's name? Anthony
How old are you? 29
What is your favorite football team? The Dallas Cowboys
How many kids do you have? 2

Hi Claire! Your husband is Anthony.
Your team is The Dallas Cowboys.
You are 29 years old and have 2 kids.
```

Another Approach for Reading Data Perhaps you have experienced this problem when you accidentally enter a non-numeric value when `cin >>` is expecting a number. Uh-oh! With these console applications, you end up with wrong data in variables, sometimes the program hangs or the screen gets stuck in an infinite scrolling pattern. When you write graphical user interfaces with combo-boxes, text fields, buttons and sliders, you are able to control the way your user interacts with your program and check the input. Using these console application windows and keyboard input, we sometimes experience unfriendly results.

The safe approach for reading numeric and character data is to use a `getline` for reading numeric values. This method is accomplished by reading the numeric value into a character string. The `atoi` function converts ASCII characters to integers. (The `atol` function converts characters to long integers.) The `atof` function converts a C-string to a double or floating point value. In C++'s standard namespace, the `atoi` and `atof` functions are available, and we don't need to include any library for their use. If you used the C header formats, you'd need to include `<stdlib.h>` for the `atoi` and `atol` functions, and `<math.h>` for `atof`. Using these functions is preferred because, if the user enters some nonnumeric information, the `atoi` and `atof` functions (in Microsoft Visual C++) return a zero value.

Special note: The `atoi` function, along with `atof`, will convert the character data into numeric data—as long as the string contains valid information. If the string contains invalid data, the return value is undefined by the ISO standard—however, many implementations, including Microsoft's Visual C++ 2005 Express, will return a zero. This may cause a problem if the program must distinguish between a real zero and an error zero. There are other string-to-number functions

atoi function

in the standard library; converts a character string into an integer

atol function

in the standard library; converts a character string into a long integer

atof function

in the math library; converts a character string into a float or double

(*strtod*, *strtol*, etc.) that offer more control over the conversion. These functions return null pointers if the data are invalid, and they are more complicated to use. These functions are also in C's standard library.

In Program 6-9 we use a character array named buffer to read the numeric values. We then convert the values in the buffer to their numeric variables.

Program 6-9

```
1 //Demonstrate how to use C-strings to read both
2 // character data and numeric data into a program.
3
4 //The input data in the C-string buffer can be
5 //converted to the desired numeric data.
6
7 #include <iostream>
8 using namespace std;
9
10 int main()
11 {
12
13     cout << "\n The How Old Are You Claire? Program Part 2"
14         << "\n Once again, ask our friend, "
15         << " Claire some personal questions.\n\n";
16
17     char husband[25], footballTeam[25];
18     //make a temporary buffer to read in numeric data
19     char buffer[25];
20     int age, numKids;
21
22     double PI;
23
24     //ask Claire for husband's name, her age,
25     //favorite football team and number of kids
26
27     cout << "\n Hi Claire, what is your husband's name? ";
28     cin.getline(husband, 25);
29
30     //Now ask age
31     cout << "\n How old are you?    ";
32     cin.getline(buffer,25);
33
34     //convert using ASCII to Integer function
35     age = atoi(buffer);
36
37     //Team name? Use another getline
38     cout << "\n What is your favorite football team?   ";
39     cin.getline(footballTeam,25);
40
41     //Number of kids
```



```
42     cout << "\n How many kids do you have?      ";
43     cin.getline(buffer,25);
44     numKids = atoi(buffer);
45
46     //Let's see if Claire knows PI's value.
47     cout << "\n What is the value of PI?      ";
48     cin.getline(buffer,25);
49
50     //convert to double with atof
51     PI = atof(buffer);
52
53     // write info out to screen
54     cout << "\n Hi Claire! Your husband is " << husband
55             << ".\n Your team is " << footballTeam <<
56             ". \n You are " << age << " years old and have "
57             << numKids << " kids. \n PI is " << PI << endl;
58
59     return 0;
60 }
```

Output

The How Old Are You Claire? Program Part 2
Once again, ask our friend, Claire some personal questions.

Hi Claire, what is your husband's name? Anthony
How old are you? 29
What is your favorite football team? Cowboys
How many kids do you have? 2
What is the value of PI? 3.14159

Hi Claire! Your husband is Anthony.
Your team is Cowboys.
You are 29 years old and have 2 kids.
PI is 3.14159.

Character String Functions Provided in C++

The C language provided the programmer several useful functions for working with character arrays. These functions provided the tools for copying, concatenation, comparison, searching, etc. Tables 6-1 illustrates the prototype and call statements for these functions. The prototypes for the character string functions show character pointers as input (*char**). Remember that C++ provides a pointer to the first element of the array for every array that is declared. When we use the array name, we are simply using the pointer to the array.

In Program 6-10 we see a few of these C-string functions in action. This program formats the user's mailing information into an address label. The exciting part of this program is the construction of the name in the label. We ask the user for first, middle, and last name, then put the name in a separate array containing last, first, and middle initial format. Filling the parts and pieces in this new Name

TABLE 6-1Functions in *cstring* Library

Function Prototype ^a and Function Name	Purpose and Example ^b
<code>void strcat(char* dest, char* src);</code> <code>strcat(s1,s2);</code>	Concatenates <i>src</i> onto <i>dest</i> . <code>char s1[30] = "I love C++";</code> <code>char s2[10] = " very much!";</code> <code>strcat(s1,s2);</code> <code>s1 now has "I love C++ very much!"</code>
<code>void strcpy(char* dest, char* src);</code> <code>strcpy(s1,s2);</code>	Copies <i>src</i> into <i>dest</i> . <code>char s1[30], s2[30] = "I love C++";</code> <code>strcpy(s1, s2);</code> <code>s1 now has "I love C+"</code>
<code>int strcmp(char* s1, char* s2);</code> <code>int n;</code> <code>n = strcmp(s1, s2);</code> <code>or</code> <code>if(strcmp(s1,"yes")==0)</code>	Compares <i>s1</i> to <i>s2</i> , returns 0 if same, <0 if <i>s1</i> < <i>s2</i> and > 0 if <i>s1</i> > <i>s2</i> . Often used in if or while statements or used to alphabetize words. <code>char s1[15] = "Apples";</code> <code>char s2[15] = "Bananas";</code> <code>if(strcmp(s1,"Apples") ==0) // checks</code> <code>to see if s1 is Apples</code> <code>if(strcmp(s1, s2) < 0) // if true, s1</code> <code>is before s2 in the alphabet</code>
<code>char * strstr(char*, char*);</code> <code>char *char_ptr;</code> <code>char_ptr = strstr(s1,s2);</code>	Returns a pointer to the first occurrence of <i>s2</i> in <i>s1</i> , or NULL if <i>s2</i> is not found. <code>char s1[50] = "The rain in Spain is</code> <code>mainly on the plain.";</code> <code>char s2[10] = "elephant"</code> <code>char s3[10] = "rain";</code> <code>char *char_ptr;</code> <code>char_ptr = strstr(s1,s2); //char_ptr</code> <code>is NULL since elephant is not in s1</code> <code>char_ptr = strstr(s1,s3); //char_ptr</code> <code>points to "r"</code>
<code>int strlen(char *);</code> <code>int length = strlen(s1);</code>	Returns an integer value length of string not including terminating null. <code>int length;</code> <code>char s1[25] = "I love C++!";</code> <code>length = strlen(s1); // length is 11</code>
<code>char* strncpy(char* dest,</code> <code> char *src,int n)</code>	Copies characters from source to destination. <code>char s1[20] = "Cats are good pets.;"</code> <code>strncpy(s1,"Dogs",4);</code> <code>s1 now has Dogs are good pets.</code>

^aPrototypes are given for reference only. The user needs to use the standard namespace.^bAssumes all strings are null-terminated.

array is a tedious task—but not difficult. Look over Table 6-1 and read through Program 6-10. The source code comments explain the process.

Program 6-10

```
1 //This program creates an label maker
2 //using C-strings and several functions from
3 //C's <string.h> These functions are now available
4 //in C++'s standard namespace
5
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     char first[20], middle[20], last[20];
12     char Name[50], Address[50], City[30];
13     char State[5], Zip[15];
14
15     //first ask for information, read using getline
16     cout << "\n Enter first name: ";
17     cin.getline(first,20);
18     cout << "\n Enter middle name: ";
19     cin.getline(middle,20);
20     cout << "\n Enter last name: ";
21     cin.getline(last,20);
22
23     cout << "\n Enter street address: ";
24     cin.getline(Address,50);
25     cout << "\n Enter city: ";
26     cin.getline(City,30);
27     cout << "\n Enter 2 letter state abbrev: ";
28     cin.getline(State,20);
29     cout << "\n Enter zip code: ";
30     cin.getline(Zip,15);
31
32     //build Name Last, First Middle Initial
33     //first, copy the last name into Name
34     strcpy(Name,last);
35
36     //next, concatenate a comma
37     //use ", " since that is considered a char[]
38     strcat(Name,", ");
39
40     //add first name
41     strcat(Name,first);
42
43     //add space
```

```

44     strcat(Name," ");
45
46     //pull first initial, use strncat
47     strncat(Name,middle,1);
48
49     //add period
50     strcat(Name,".");
51
52     cout << "\n Here is the mailing label.\n\n"
53             << Name << endl << Address << endl
54             << City << ", " << State << "    " << Zip << endl;
55
56     return 0;
57 }
}

```

Output

Enter first name: Robert
 Enter middle name: Andrew
 Enter last name: Thomas
 Enter street address: 4321 33rd Circle
 Enter city: Phoenix
 Enter 2 letter state abbrev: AZ
 Enter zip code: 83383

Here is the mailing label.
 Thomas, Robert A.
 4321 33rd Circle
 Phoenix, AZ 83383.

6.5 Multidimensional Arrays

multidimensional array

an array that represents a table (rows and columns) or a layered table

It is possible to declare a ***multidimensional array*** (an array that represents a table with rows and columns) in C++. The declaration for a two-dimensional array is:

dataType arrayName [number of rows][number of columns];

such as:

```

int grid[3][4];           //this grid has 3 rows and 4 columns
int image[640][480];      //this image has 640 rows and 480 columns

```

The only difference between a one-dimensional array and a two-dimensional array is that the two-dimensional array has a second size specification within a second

set of brackets. A three-dimensional array is declared with rows, columns, and height (or layers):

```
dataType arrayName [number of rows][number of columns][number of layers];
```

Here is an example of a 3-D array containing geographic information over time.

```
int weeklyWeatherGrid [256][256][7];
```

Figure 6-10 illustrates one-, two-, and three-dimensional array declarations and the manner in which the elements of the array can be visualized.

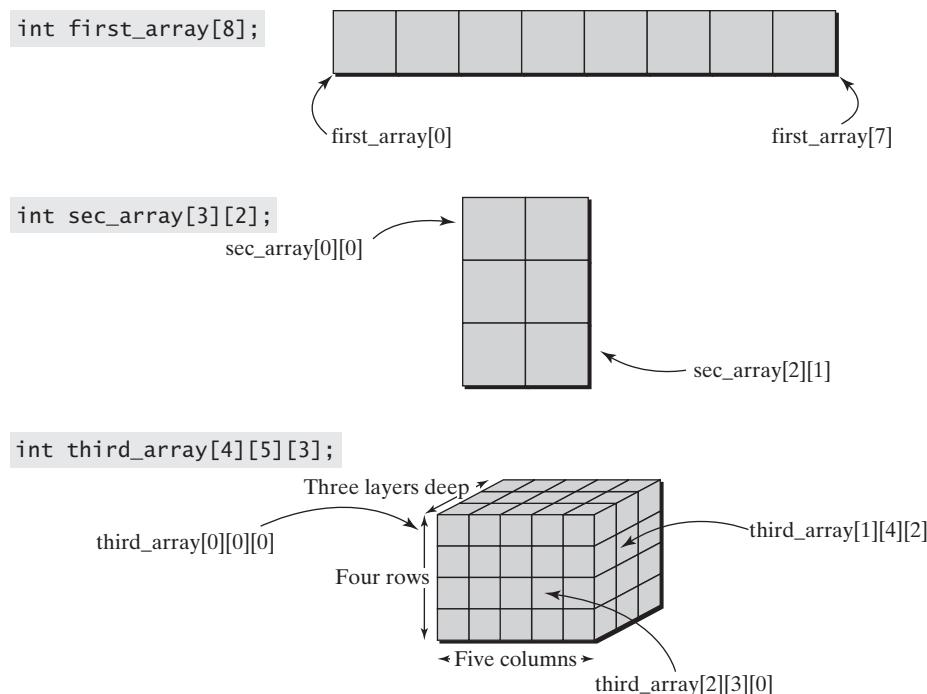


Figure 6-10
One-, two-, and
three-dimensional
arrays.

Two-Dimensional Array Initialization

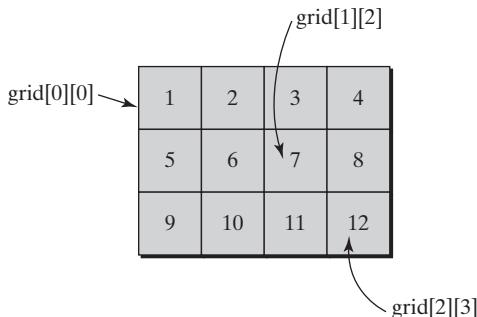
Two-dimensional arrays can be initialized when declared, and C++ fills the rows of the array first. Figure 6-11 shows a two-dimensional array that is three rows by four columns. To initialize this array, the following statement is required:

```
int grid[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Another way to write it (to make it easier to visualize) is:

```
int grid[3][4] = { 1, 2, 3, 4,
                  5, 6, 7, 8,
                  9, 10, 11, 12 };
```

```
int main()
{
    int grid[3][4] = { 1,2,3,4,5,6,7,8,9,10,11,12};
```

**Figure 6-11**

Initializing two-dimensional arrays.

Nested *for* Loops and Two-Dimensional Arrays

Two-dimensional arrays require two indexes, and typically they need to vary. Nested *for* loops offer an easy way to set up and keep track of these indexes. You may name the integer index variables appropriately so that the code is easy to read.

If we set up a two-dimensional array of float values called *grid* that is 100 rows by 50 columns, nested *for* loops will be used to access each array variable and to place zero in each element. In the code below, this set of nested *for* loops will traverse the array, moving across each row.

```
float grid [100][50];           //2D array of 100 rows by 50 columns
int row, col;
for(row = 0 ; row < 100; ++row)      //outer loop
{
    for(col = 0; col < 50; ++col)    //inner loop
    {
        grid[row][col] = 0.0;
    }
}
```

If we take a closer look at the assignment statement in the inner part of the *for* loops, we see that this statement will be executed a total of 5000 times. Each time the program reaches this statement, row and column indices are different. Figure 6-12 shows the exact order in which the *grid* elements are accessed.

BINGO with a Two-Dimensional Array

Did you ever play Bingo at school? Bingo is a game of luck and concentration where randomly selected numbers are drawn and matched on your Bingo card. Your Bingo card is 5 by 5 with a unique random number in each square. The numbers range from 1 to 75. In the game of Bingo, there is a system (or person) that generates a letter-number combination, then calls it out to the players. If you have

The inner loop statement is executed 5,000 times.

```
float grid [100][50];
int row, col;

for(row = 0 ; row < 100; ++ row) // outer loop
{
    for(col = 0; col < 50; ++ col) // inner loop
    {
        grid[row][col] = 0.0;
    }
}
```

grid[0][0]	1st time
grid[0][1]	2nd
grid[0][2]	3rd
:	
grid[0][49]	50th
grid[1][0]	51st
grid[1][1]	52nd
grid[1][2]	53rd
:	
grid[1][49]	100th
:	
grid[99][0]	4951st
grid[99][1]	4952nd
:	
grid[99][49]	5000th

Figure 6-12
Accessing elements 100×50 two-dimensional array.

that letter-number on your card, you mark it. The goal of Bingo is to mark the squares on your card so that a winning pattern is created. (Winning patterns can vary from all in a row or column, to diagonals, to outer edges, to full cards.) Once you have a winning pattern you get to yell, “BINGO!”

In Program 6-11 we use the random number generator to help us build a Bingo card. What is important here is that we use a two-dimensional array for our card. We use nested *for* loops for both filling and writing the array. Study the code and note how we use an offset value for the value in each column. The *continue* statement comes in handy in our nested loop—if we are at the center square, we place 100 in the square (indicating it is the FREE square). We then continue on to the next loop iteration. Oh, there is another problem! There are duplicate values on the Bingo card! Seems like a revision of this program will be necessary, don’t you think?

Program 6-11

```
1 //This program fills a BINGO card using
2 //the rand() function.
3 //We do not check for duplicates on the card. :-(

4
5 //The card is represented by a 5x5 int array.

7 //BINGO cards have a range of 75 random numbers
8 // First Col B 1-15
9 //Second Col I 16-30
10 // Third Col N 31-45
11 //Fourth Col G 46-60
12 // Fifth Col O 61-75
13
14 #include <iostream>
```



```
15 #include <iomanip>
16 using namespace std;
17
18 int main()
19 {
20     //use 2D array of ints for our card
21     int BingoCard[5][5];
22
23     cout <<"\n We're going to generate a BINGO card"
24         << "\n using the random number generator. \n";
25
26     //We'll fill the card column by column.
27
28     int row, col, value;
29
30     for(col = 0; col < 5; ++ col) //work on each col
31     {
32         //now go down each row
33         for(row = 0; row < 5; ++ row)
34         {
35             //check if we're at [2][2] FREE
36             if(row == 2 && col == 2)
37             {
38                 BingoCard[row][col] = 100;
39                 continue; //jump to next loop iteration
40             }
41
42             //first generate a number between 1-15
43             value = rand()%15 + 1;
44
45             //offset it by 15 for each column
46             BingoCard[row][col] = value + (col*15);
47         }
48     }
49
50     cout << "\n Here is your BINGO card. Good Luck! \n";
51     char Top[10] = "BINGO";
52
53     for(int i = 0; i < 5; ++i)
54         cout << setw(6) << Top[i];
55
56     cout << endl;
57
58     //now print the card row by row
59     //use nested loops again
60     for(row = 0; row < 5; ++ row)
61     {
62         for(col = 0; col < 5; ++ col)
```

```
63         {
64             if(col == 2 && row == 2)
65                 cout << setw(6) << "FREE";
66             else
67                 cout << setw(6) << BingoCard[row][col];
68         }
69         cout << endl;
70     }
71
72     return 0;
73 }
```

Output

We're going to generate a BINGO card

using the random number generator.

Here is your BINGO card. Good Luck!

B	I	N	G	O
12	20	36	47	67
3	19	36	57	67
5	19	FREE	56	70
11	23	32	48	63
15	30	43	58	64

This output looks good, except for the pesky problem of duplicate values (i.e., two I-19s). Hmm? How would you write this program so you do not have duplicate values? Read on—and we'll take care that problem soon!

6.6

Multidimensional Arrays and Functions

When passing an array (reference) to a function, two-, three-, and higher dimensional arrays require that the programmer specify all but the beginning array size in the prototype and function header line. When you declare a multidimensional array, C++ multiples the dimensions together and reserves that many elements for the array. For example, if you declared these two arrays:

```
int puzzle[16][10]; // puzzle has 16*10 = 160 total elements
int test[4][40]; // test has 4*40 = 160 total elements
```

C++ reserves 160 integers for both sets of data. The array reference that the function receives points to the first element—but in order for a function to know how to interpret the data correctly, we need to give it enough information so that it knows the number of rows and columns. For example, if a program has a two-dimensional array, like this:

```
int image[320][240]; // image has 320*240 = 76800 total elements
```

We have a function work with this data, the function prototype and function header line need to know that the array reference is for a 2-D array, as well as how the data is organized. The prototype for our make-believe *ImageFunction* is:

```
// function prototype includes last dimension's size  
void ImageFunction(int image [][]240);
```

It is then business as usual inside the function.

```
// function body uses  
void ImageFunction(int image [][]240) {  
    //now just work on image[i][j] elements  
}
```

An image from a medical Magnetic Resonance Imaging machine (MRI) usually consists of individual slices through the body, stacked together to give medical professionals a three-dimensional view of the body. Often these stacks are placed into a three-dimensional array and manipulated by computer programs to view various parts in the scan. The three-dimensional array that holds all the MRI image data could be declared like this:

```
int MRISeries[256][256][25]; // holds 25 slices of an MRI scan, 256x256
```

Perhaps we had a function to view the 3-D data. The prototype would need to have the last two dimensions:

```
// 3-D array needs last two dimension sizes  
void ViewMRIScan(int MRISeries [][]256[25]);
```

Inside the function, there are three array indices for our data.

```
// function body uses  
void ViewMRIScan(int MRISeries [][]256[25]) {  
    //now just work on MRISeries[i][j][k] elements  
}
```

Revisit Bingo Card Program

The previous Bingo program had the entire code in the *main* function, so of course we are going to rewrite it using functions. Program 6-12 produces a Bingo card with no duplicate values on the card. The program is designed to use two functions, *FillCard* and *PrintCard*. Both functions are passed the *BingoCard* array. Notice how the function prototypes and function header lines contain the *[] [5]* notation, indicating that it is a two-dimensional array.

Now is an excellent time to fix the duplicate values problem that is a result from our random number generating giving us repeated values. Here we use an array of boolean values (true/false) to “remember” whether a value has been used on the Bingo card. The *FillCard* algorithm for avoiding duplicate values is this: because there are 75 possible values on the Bingo card, we use a boolean

1st Fill check array
with falses

false
false
false
false
.
.
false
check[0]
check[1]
check[2]
check[3]
.
check[75]

Next generate a random value

```
value = rand()%15 + 1
value = value + (col * 1);
(Assume value is 3, col is 0)
```

if check[value] is false, we haven't used that value yet.
Place value in card and set check[value] to true.

false
false
false
true
.
.

B	I	N	G	O
3				

Figure 6-13

The check array “remembers” if we have used a value on our Bingo card.

array named *check*, that is sized to 76 (elements 0–75). Each element represents the status of a value for the Bingo card. (The *check* array is so named because we use it to check if values have been used.) We generate a value for the Bingo card—if the *check* element at that value is false, it means that we haven’t used that number on the card. We then set the *check[value]* to true. Figure 6-13 illustrates this technique. Study Program 6-12. It is full of juicy programming information.

Program 6-12

```
1 //This program fills a BINGO card using
2 //the rand() function.
3
4 //We use several of our own functions to help us.
5
6 #include <iostream>
7 #include <iomanip>
8 using namespace std;
9
10 //Function prototypes
11 void PrintCard(int BCard[] [5]);
12 void FillCard(int BCard[] [5]);
13
14 int main()
15 {
16     //use 2-D array of ints for our card
17     int BingoCard[5] [5];
18
19     cout << "\n We're going to generate a BINGO card"
```

```
20      << "\n using the random number generator. \n"
21      << "\n No duplicate values allowed on our card!\n";
22
23      FillCard(BingoCard);
24
25      PrintCard(BingoCard);
26
27      return 0;
28  }
29
30 /*This card fills the Bingo card with random numbers.
31    BINGO cards have 75 random numbers
32    First Col B 1-15
33    Second Col I 16-30
34    Third Col N 31-45
35    Fourth Col G 46-60
36    Fifth Col O 61-75
37
38    We also use a bool check array to keep track
39    of generated values. If we have used a number
40    we have to regenerate one for that square.
41    This prevents duplicate values on our card. */
42
43 void FillCard(int BCard[][]) {
44
45     int row, col, value, i;
46     bool check[76]; //need values 1-75
47
48     //First fill check array with falses
49     for(i = 0; i < 76; ++i)
50         check[i] = false;
51
52     bool again; //flag to get another value
53
54     for(col = 0; col < 5; ++ col) //work on each col
55     {
56         //now go down each row
57         for(row = 0; row < 5; ++ row)
58         {
59             //check if we're at [2][2] FREE
60             if(row == 2 && col == 2)
61             {
62                 BCard[row][col] = 100; //indicate FREE
63                 continue; //jump to next loop iteration
64             }
65
66             again = true;
67             do
```



```
68         {
69             //first generate a number between 1-15
70             value = rand()%15 + 1;
71
72             //offset it by 15 for each column
73             value = value + (col*15);
74
75             //now check if we've gotten that number before
76             //if we have, do this again
77             if(check[value] == false)
78             {
79                 BCard[row][col] = value;
80                 check[value] = true;
81                 again = false;
82             }
83         }while(again == true);
84     }
85 }
86 }
87
88 void PrintCard(int BCard[][5])
89 {
90     cout << "\n Here is your BINGO card. Good Luck! \n";
91     char Top[10] = "BINGO";
92     int row, col, i;
93
94     for(i = 0; i < 5; ++i)
95         cout << setw(6) << Top[i];
96
97     cout << endl;
98
99     //now print the card, row by row
100    for(row = 0; row < 5; ++ row)
101    {
102        for(col = 0; col < 5; ++ col)
103        {
104            if(col == 2 && row == 2)
105                cout << setw(6) << "FREE";
106            else
107                cout << setw(6) << BCard[row][col];
108        }
109        cout << endl;
110    }
111 }
```

Output

We're going to generate a BINGO card
using the random number generator.

No duplicate values allowed on our card!

Here is your BINGO card. Good Luck!

B	I	N	G	O
12	20	36	56	63
3	19	32	48	64
5	23	FREE	58	68
11	30	43	52	67
15	21	42	55	72

Snow White: Two-Dimensional Array of Names Program

Snow White's software can use an array to keep track of the dwarfs' names. Using a two-dimensional array of character arrays in C++ is one way to maintain a list of text-type variables such as names or labels. One technique for initializing a list of names involves declaration and initialization in one statement. The code in Program 6-13 sets up an array for Snow White that will initialize the dwarfs' names and then call a function to print the names to the screen. Figure 6-14 shows the contents of the dwarfs array. Notice how C++ has filled the remainder of each row with the null character.

Program 6-13

```

1 //A short program with Snow White and The Seven Dwarfs
2 //Uses a 2-D C-string Array.
3
4 #include <iostream>
5 using namespace std;
6
7 //prototype shows input is a 2-D array
8 void WriteNames(char dwarfs[])[10];
9
10 int main()
11 {
12     // Declare and initialize the array
13     char dwarfs[7][10] = { "Doc", "Grumpy",
14                           "Dopey", "Bashful", "Sleepy",
15                           "Sneezy", "Happy" };
16

```

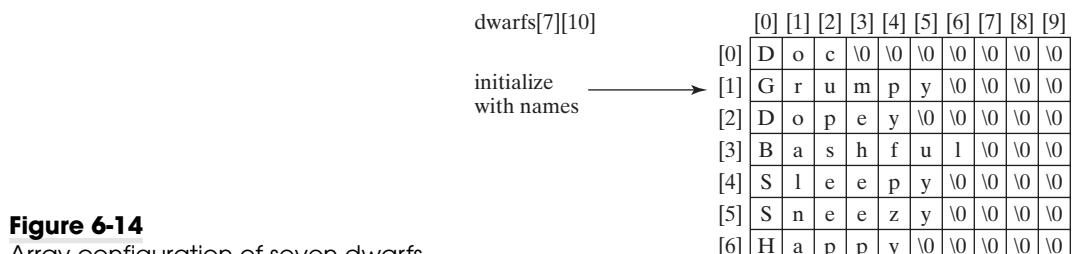


Figure 6-14

Array configuration of seven dwarfs.

```
17     WriteNames(dwarfs);
18
19     return 0;
20 }
21
22 //function to write the seven names
23 void WriteNames(char dwarfs[][][10])
24 {
25     cout << "\n Here are the Seven Dwarfs \n";
26     for (int i = 0; i < 7; ++i)
27     {
28         cout << " Dwarf # " << i+1 << " is " << dwarfs[i] << endl;
29     }
30 }
```

Output

```
Here are the Seven Dwarfs
Dwarf # 1 is Doc
Dwarf # 2 is Grumpy
Dwarf # 3 is Dopey
Dwarf # 4 is Bashful
Dwarf # 5 is Sleepy
Dwarf # 6 is Sneezy
Dwarf # 7 is Happy
```

6.7

Filling Arrays from Data Files

It is not uncommon to find that our arrays contain a large number of values. To illustrate and practice using numeric arrays, it is impractical to write programs asking the user to enter the data from the keyboard. (If you have been running the different versions of the Phone Bills program, you know how inconvenient it is to enter the 12 bill values by hand.) In real programs, data are usually read from a data file (or files), or obtained from a database. C++ provides powerful file input and output tools. There are several ways to read from and write to files. Because we are now working with arrays, let's see a few techniques for reading text (both character and numeric) from a data file. We also write data to an output file.

Using data files in C++ does not require magic, as some programmers may claim. The program needs to be told where the data file is located and it needs to open the file. Once open, the data can be read from the file. We must close the file when we are finished reading the data. This step should be performed once all of the data has been read—which may be at the end of a function, or at the end of the program.

One last feature when working with data files is that the programmer must know exactly how the data is formatted (written) in the data file. The read statements must correspond to the way the data is formatted in the file. Appendix F, “File Input/ Output,” contains more examples for reading textual and binary data as well as writing data to an output file. If you haven’t already, read Appendix F.

More Fun with C++ Classes: the *ifstream* and *ofstream* Classes

We see the C++ streams in action when we write to the screen using *cout* and read from the keyboard using *cin* and *getline*. We also use the *stringstream* class to create objects that give us formatted strings. Now we learn about two more stream classes that allow us to read from, and write to, data files. In C++, whether program data is coming from the keyboard or a file, and going to the console window or a file, the general mechanics are the same. Now that you are expert with *cout*, *cin*, *getline*, and *stringstream*, you are nearly an expert with file input/output too!

ifstream class

in the *stream* library;
contains functions for
reading data from a
file into a program

ofstream class

in the *stream* library;
contains functions for
writing data from a
program to a data
file

The class ***ifstream*** contains the necessary functions for opening and reading data from files. The class ***ofstream*** contains the functions for writing data to a file. Because they are stream classes, many of the things we do when using *cout* or *cin* apply directly to our *ifstream* and *ofstream* objects.

The Phone Bill Program with Data Files For the last time, let's rewrite and expand the Phone Bills program so that we read the monthly values from a data file. We'll then calculate the yearly total, the average monthly bill, find the highest bill, and write the results data to a file. Figure 6-15 shows the data file formats as well as the general outline of the main function. We'll have separate functions perform program tasks, including one function that reads the file and fills the array,

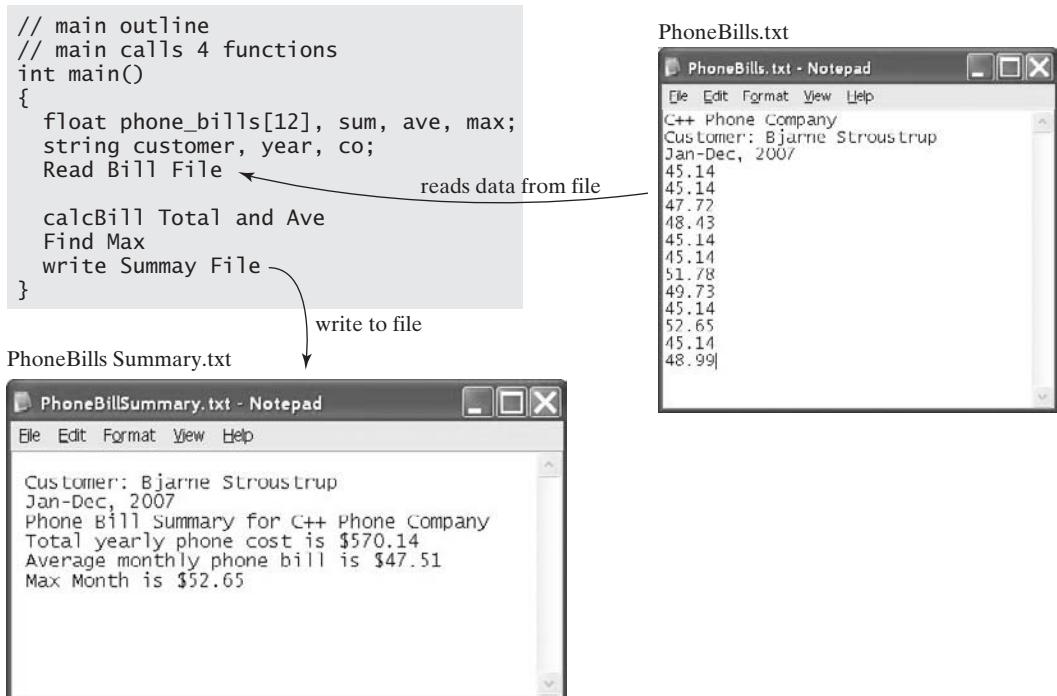


Figure 6-15

Phone bills program with data files.

one that calculates the sum and average, one that finds the max value, and one that writes the results to an output file.

Program 6-14 is a bit longer, and hence has been separated into three source files. We'll concentrate on each file and then look at the output. First, let's examine the PhoneFunctions.h file. It contains the prototypes for the four functions that our main function will be calling. The *ReadBillData* function's job is to read the file, fill the array and return all pertinent data to main. Reference variables are used for the three strings. The *CalcBillTotalandAve* is from our Phone Bills and Function program (6-5). We include only what this file needs—which is the <string> library.

Program 6-14

```
1 //File: PhoneFunctions.h
2
3 #include <string>    //needed here for string
4 using namespace std;
5
6
7 //Function prototypes
8 bool ReadBillData(float bills[], string &rCust,
9                     string &rCompany, string &rYear);
10
11 float CalcBillTotalandAve(float bills[], float &rSum);
12
13 float FindMax(float bills[]);
14
15 bool WriteSummaryFile(string cust, string year, float sum,
16                         float ave, float max);
```

The main function is contained in the file PhoneBills.cpp. It is straightforward, calling the functions to complete the purpose of this program. One thing to note is that the *Read* and *Write* functions return boolean values. These true/false values indicate if the function encountered a problem while opening the data files. Some people believe it is OK to just pop up an error message and exit the program in the function if there is a file problem. This is not a good habit to start! If you are part of a team of programmers who are building a large program, do not have a function terminate the program. Send back an error flag and let the calling function determine its fate. Here, the calling function is *main*. We've chosen to use the *exit()* function inside *main* to terminate the program if there is a problem with the file. It is safe to have your program terminate in *main* as it is the program's driver. The program output to the console window is shown below *main*.



Program 6-14

```
1 //A program that reads the monthly bills
2 //from a data file. It calculates phone bill
3 //average and total values for the year.
4
```

```
5  #include <iostream>
6  #include <string>
7  using namespace std;
8
9  #include "PhoneFunctions.h"
10
11 int main()
12 {
13     string customer, company, year;
14     float phone_bills[12];
15
16     cout << "\n A program that determines yearly total"
17         << "\n and average monthly phone bill."
18         << "\n\n Data contained in \"PhoneBills.txt\""
19         << "\n Results written to \"PhoneBillSummary.txt\"\n";
20
21     //Call function to read data into phone_bills
22     //Function returns a false if it can't find file.
23     bool result = ReadBillData(phone_bills, customer,
24                                company, year);
25     if(result == false)
26     {
27         cout << "\n Can't continue working, no data!";
28
29         //No file to read, so terminate the program.
30         exit(1);
31     }
32
33     //Calc values
34     float ave, sum, max;
35     ave = CalcBillTotalandAve(phone_bills, sum);
36
37     //Find the max bill
38     max = FindMax(phone_bills);
39
40     result = WriteSummaryFile(customer, year, sum, ave, max);
41     if(result == false)
42     {
43         cout << "\n Problem writing the file.";
44     }
45     else
46     {
47         cout << "\n All done! "
48             << "\n Check \"PhoneBillSummary.txt\" for results!" << endl;
49     }
50     return 0;
51 }
52 }
```



Output

A program that determines yearly total
and average monthly phone bill.

```
Data contained in "PhoneBills.txt"  
Results written to "PhoneBillSummary.txt"  
  
All done!  
Check "PhoneBillSummary.txt" for results!
```

Our third source file contains the best part of this program! There is one function that reads the data file and one that writes the information to the data file. (Once again, refer to Figure 6-15 to see the files.) To make our life easier, we use `#define` statements to identify the names of the files. We place these lines at the top of the file.

```
//For convenience, we'll use #define for file names  
//We place these lines outside of the functions, at the top of the file.  
#define FILE_IN    "PhoneBills.txt"  
#define FILE_OUT   "PhoneBillSummary.txt"
```

We'll look at reading the file first. Inside the `ReadBillData` function we must make an `ifstream` object, then tell it the name of the file that it will read. After the `open` statement we check that the file was opened (see `if` statement).

```
ifstream billInput;      //this is the input stream object  
billInput.open(FILE_IN); //here we open it  
                      //FILE_IN is defined to be "PhoneBills.txt"  
//check that we've got the file open  
if(!billInput)  
{  
    cout << "\n Whoops! Can't find " << FILE_IN;  
    return false;  
}
```

If there are any problems finding or opening the file, the program writes the “Whoops” message and returned a false. No problems, then, we are ready to start reading. We use the `billInput` object just as we would the `cin` object. (See code below.) We read the first three lines to obtain the company, customer, and year data. We then use a `for` loop to read the bill values. The output file is the same idea. We make an `ofstream` object, open it, check that it was opened without problems, then write data to the file.

Finding Array Maximum (or Minimum) To determine the maximum value in the array, we need to search the array and compare each value to a maximum variable. If we find an array value higher than the maximum, we place that value in the `max` variable. You may ask yourself, “What value do we put in `max` to get started?”

There are two techniques available to us for setting the initial value of `max`. The first *and not recommended* approach, involves trying to guess a really low value that we know is smaller than the expected highest value in the array and use it. For example, you may guess that the lowest phone bill might be zero, but what if

you had a credit one month? Then zero would not work! The second approach is to set the max value to the first element in the array and use it as an initial comparison value. This technique guarantees that the program always uses a number within range and that the maximum will be determined. We use this technique here and in other programs in the Practice section.

Examine the functions below, and read the comments for more information. The files we see in Figure 6-15 were generated by this program.

Program 6-14

```
1  //File: PhoneFunctions.cpp
2
3  #include <iostream>           //needed for cout
4  #include <string>            //needed for string
5  #include <fstream>           //for ifstream and ofstream
6  using namespace std;
7
8
9  //For convenience, we'll use #define for file names
10 #define FILE_IN "PhoneBills.txt"
11 #define FILE_OUT "PhoneBillSummary.txt"
12
13 //Function that reads monthly bill data.
14 bool ReadBillData(float bills[], string &rCust,
15                     string &rCompany, string &rYear)
16 {
17
18     ifstream billInput;           //input object
19
20     billInput.open(FILE_IN);
21
22     //check that we've got the file open
23     if(!billInput)
24     {
25         cout << "\n Whoops! Can't find " << FILE_IN;
26         return false;
27     }
28
29     //Read monthly bill info using ifstream object
30     //getline and cin works just like it did
31     //for keyboard input--except use ifstream object
32
33     //First line is the Company
34     getline(billInput,rCompany);
35
36     //Second line is the Customer
37     getline(billInput,rCust);
38
39     //Third line is the timeframe
```

```
40     getline(billInput,rYear);
41
42
43     //Used this way, billInput works just like cin >>
44     for(int i = 0; i < 12; ++i)
45     {
46         billInput >> bills[i];
47     }
48
49     //all done reading, close the file
50     billInput.close();
51
52     return true;
53 }
54
55 //Function that determines average phone bill
56 //given the yearly sum.
57 float CalcBillTotalandAve(float bills[], float &rSum)
58 {
59     rSum = 0.0;
60     for(int i = 0; i < 12; ++i)
61     {
62         rSum = rSum + bills[i];
63     }
64
65     return rSum/12.0;
66 }
67
68 float FindMax(float bills[])
69 {
70     //First set max to first element, then compare
71     //to all other values. If we find a higher value,
72     //set it to max;
73
74     float max = bills[0];
75
76     for(int i = 1; i < 12; ++i)
77     {
78         if(max < bills[i])
79             max = bills[i];
80     }
81
82     return max;
83 }
84
85 bool WriteSummaryFile(string cust, string year, float sum,
86                         float ave,float max)
87 {
```

```

88     ofstream billOutput;           //output object
89
90     billOutput.open(FILE_OUT);
91
92     //check that we've got the file open
93     if(!billOutput)
94     {
95         cout << "\n Whoops! Can't open " << FILE_OUT;
96         return false;
97     }
98
99     //Now it is just like cout << to write to the file!
100    billOutput << "\n " << cust << "\n " << year;
101    billOutput << "\n Phone Bill Summary";
102
103   //Set precision for the file stream object
104   billOutput.precision(2);
105   billOutput.setf(ios::fixed);
106   billOutput << "\n Total yearly phone cost is $" << sum
107           << "\n Average monthly phone bill is $"
108           << ave << "\n Max Month is $" << max << endl;
109
110   //close the file
111   billOutput.close();
112
113   return true;
114 }
```

Writing and reading data to and from files is a snap with the *ofstream* and *ifstream* objects! You know how to write to the console window and read from the keyboard, it is a short step to be able to write/read files. All the same rules apply! We'll see two more file input/output programs in the Practice section of this chapter, and in Appendix F. (Remember, F is for files.)

6.8

Summary

Arrays organize and represent data in a C++ program. Key array concepts are presented here:

- An array represents a group of variables that can be thought of as a list of values. This list (or group) is referred to with one name, and each individual element is referenced using an integer index value.
- The array can be set up as a single list or column (one-dimensional), a table (two-dimensional), a stacked table (three-dimensional), and so forth.
- When an array is declared, a pointer of the same name is created automatically. This pointer contains the address of the zero element of the array.

- An array is passed to a function by using the name of the array in the call statement. C++ actually passes a reference containing the address of the first element of the array to the function. The called function accesses the calling function's array. The called function does not have its own copy.
- A function cannot create its own local array and then return it to the calling program.

An Overview of Common Errors Encountered while Writing Arrays

C++ programmers need to be aware of the potential problems encountered while working with arrays. When an array boundary is exceeded, other program data can be corrupted or the program crashes. Take time to review the Array Out of Bounds problem discussed earlier in the chapter, and be sure you understand it. There are a few errors that programmers see when writing arrays. We'll examine them here.



Compiler Error: Cannot Convert Parameter 1 from ‘float (12)’ to ‘float’

The “cannot convert” error is a classic, seen by beginning and expert programmers alike. As we have seen previously, the “cannot convert” is always at the function call line. What you are attempting to pass to the function doesn’t match the function header line. In our Phone Bills with Files program (Program 6-14) our *ReadBillData* prototype has been incorrectly written, leaving off the *[]*:

```
bool ReadBillData(float bills, string &rCust,      //<<< ERROR HERE
                  string &rCompany, string &rYear);
```

This error is generated at the call statement in main because we are trying to pass a float array of size 12 to a single float variable.

```
float phone_bills[12];
//later in main we call ReadBillData
bool result = ReadBillData(phone_bills, customer,
                           company, year);
```

Compiler Error: Cannot Convert Parameter 1 from ‘int (100)’ to ‘int’

Here’s another rendition of the “cannot convert” error, this time generated from the Sorting Random Numbers program (Program 6-6). Sometimes when we are writing programs, we forget the parameter order that we must use when passing values to a function. The “cannot convert” error is a quick way to tell us our parameter order is incorrect. Here we mix up total and numbers passed to the *Sort* function. Look at the prototype here, first a single *int*, then the array:

```
void Sort(int total, int numbers[]);
```

This error is generated at the call statement in main because the parameters are passed in the wrong order.

```
int numbers[100], total = 100;
//later in main we call the Sort function
Sort(numbers, total);           << ERROR HERE
```

Link Error: Unresolved External Symbol “void __cdecl WriteNames(char (* const)[10])”

We cannot leave out the link errors that look daunting, especially when an array is involved. Here's an error that is generated because the prototype and function header line do not match. Program 6-13, The Snow White program has a function that works on a two-dimensional array, sized to be 7 rows by 10 columns. If we accidentally mix up our array indices in the prototype and function header line, the linker will complain. Here is the correct prototype for our Snow White program. The program sees the prototype and is expecting an array with 10 columns.

```
void WriteNames(char dwarfs[][10]);
```

The function header line shows the two-dimensional array with 7 columns. The linker can't find the *WriteName* function that matches the prototype, and reports the above error.

```
void WriteNames(char dwarfs[][7])      << ERROR HERE, SHOULD BE [] [10]
{
    //function body here
}
```

As stated in previous chapters, many compiler and linker errors are the result of a mismatch between function prototypes, function calls, and function header lines. “Cannot convert parameter” and “unresolved external symbol” errors fall under this generalization. When in doubt, check these statements and you'll be on your way to having code ready to run!

6.9 **Practice!**

Favorite Word—Two Ways

The first practice program in this chapter asks the user to enter his favorite word and then a phrase. The program calls the *FindFavorite* function that searches the phrase for the word, returning either a true or false. Because we can solve this problem using either C-string arrays or C++ string objects, instead of choosing one of them, we'll just do it both ways!

The first program has C-string arrays for the phrase, word and user's answer to “do another?” The *FindFavorite* function uses the *strstr* function to tell us if the word is in the phrase. We pass the phrase and word to *strstr*; it returns a null character if the word is not in the phrase. If the word is in the phrase, the pointer

points to the first occurrence. Because we do not need to know where the word is located (if it is in the string), we need only to check the pointer to see if it contains a null value. Last, we ask the user to enter yes or no then use the *strcmp* function to compare what the user enters with the word *yes*. If the user has entered a “yes,” *strcmp* returns a zero value.

Program 6-15

```
1 //Favorite Word Program using C-strings
2 //Ask the user for a word and a phrase
3 //and determine if the word is in the phrase.
4
5 //This version uses C-string arrays and the
6 //strstr and strcmp functions.
7
8 #include <iostream>
9 #include <string>
10 using namespace std;
11
12 bool FindFavorite(char phrase[], char word[]);
13
14 int main()
15 {
16     char phrase[60], word[20], answer[5];
17
18     do
19     {
20         cout << "\n Enter your favorite word ==> ";
21         cin.getline(word,20);
22
23         cout << "\n\n Enter a phrase ==> ";
24         cin.getline(phrase,60);
25
26         bool find_it = FindFavorite(phrase,word);
27
28         if(find_it == true)
29             cout << "\n I see \" " << word << "\" :-)\n";
30         else
31             cout << "\n I don't see \" " << word << "\" :-(\n";
32
33         cout << "\n\n Do it again?   Enter yes or no ==> ";
34         cin.getline(answer,5);
35
36         //strcmp returns 0 if C-strings match
37     }while(strcmp(answer,"yes") == 0);
38
39     return 0;
40 }
41 bool FindFavorite(char phrase[], char word[])
```

```

42  {
43      bool result;
44      char* IsItThere;      //declare a character pointer for strstr
45
46      IsItThere = strstr(phrase,word);
47
48      if(IsItThere == NULL)
49          result = false;
50      else
51          result = true;
52
53      return result;
54  }
55
}

```

Output

```

Enter your favorite word ===> bananas
Enter a phrase ==> buy bananas at the store

I see "bananas" :-
Do it again? Enter yes or no ==> yes

Enter your favorite word ===> spinach
Enter a phrase ==> veggies are good for you

I don't see "spinach" :(
Do it again? Enter yes or no ==> no.

```

In this second solution to the problem, we use the C++ string objects. Remember that the string objects come equipped with their own methods (functions) that are accessible to us by using the object name and dot operator. In the *FindFavorite* function we ask the *phrase* object to find the word. When we use the *find* method with the object, we are automatically searching that object's data. (It is like asking you if you have any quarters in your pocket. You automatically look in your pockets, not someone else's pocket!) Also, the string's *find* function will return the location of the word in the phrase. It returns a *-1* if it does not find the word.

Program 6-16

```

1  //Favorite Word Program using C++ strings.
2  //Ask the user for a word and a phrase
3  //and determine if the word is in the phrase.
4
5  //This version uses C++ string objects and the
6  //string class' find function.
7

```

```
8 #include <iostream>
9 #include <string>
10 using namespace std;
11
12 bool FindFavorite(string phrase, string word);
13
14 int main( )
15 {
16     string phrase,word, answer;
17
18     do
19     {
20         cout << "\n Enter your favorite word ==> ";
21         getline(cin,word);
22
23         cout << "\n\n Enter a phrase ==> ";
24         getline(cin, phrase);
25
26         bool find_it = FindFavorite(phrase,word);
27
28         if(find_it == true)
29             cout << "\n I see \" " << word << "\" :-) \n";
30         else
31             cout << "\n I don't see \" " << word << "\" :-(\n";
32
33         cout << "\n\n Do it again?   Enter yes or no ==> ";
34         getline(cin, answer);
35         cin.ignore();
36
37         //can just compare using == operator
38     }while(answer == "yes");
39     return 0;
40 }
41
42 bool FindFavorite(string phrase, string word)
43 {
44     bool result;
45     int WhereIsIt;           //find searches the phrase,
46                           //returns location if in phrase
47                           // -1 if not there
48
49     WhereIsIt = phrase.find(word);
50
51     if(WhereIsIt == -1)
52         result = false;
53     else
54         result = true;
55 }
```



```

56         return result;
57     }

```

The Seven Dwarfs and Files

You may ask why would anyone bother using C-string arrays in a C++ program when the C++ string objects are so much easier to use? Good, and obvious question. Here is a question for you. What if you had to manipulate the names of the seven dwarfs so that they were printed sideways or backwards on the screen? The answer to this question lies in the ability to manipulate the individual letters in a grid that the two-dimensional C-string array gives us.

Program 6-17 reads the names of the seven dwarfs into a two-dimensional character array. There are two other arrays used in the program as well. One array will hold the dwarf names backwards, and the other will contain the names in a vertical fashion. Figure 6-16 shows the relationship between the dwarfs grid and backwardsDwarfs grid. Figure 6-17 shows the dwarfs and verticalDwarfs grid relationship. The program has been split into three source code files so that we can examine the code easily. First we look at the *main* function that is shown here. We declare our three arrays, read the file, then call the vertical and reverse functions. Once again, we have designed the read function to return a boolean value so we will know if there is a problem reading the file.

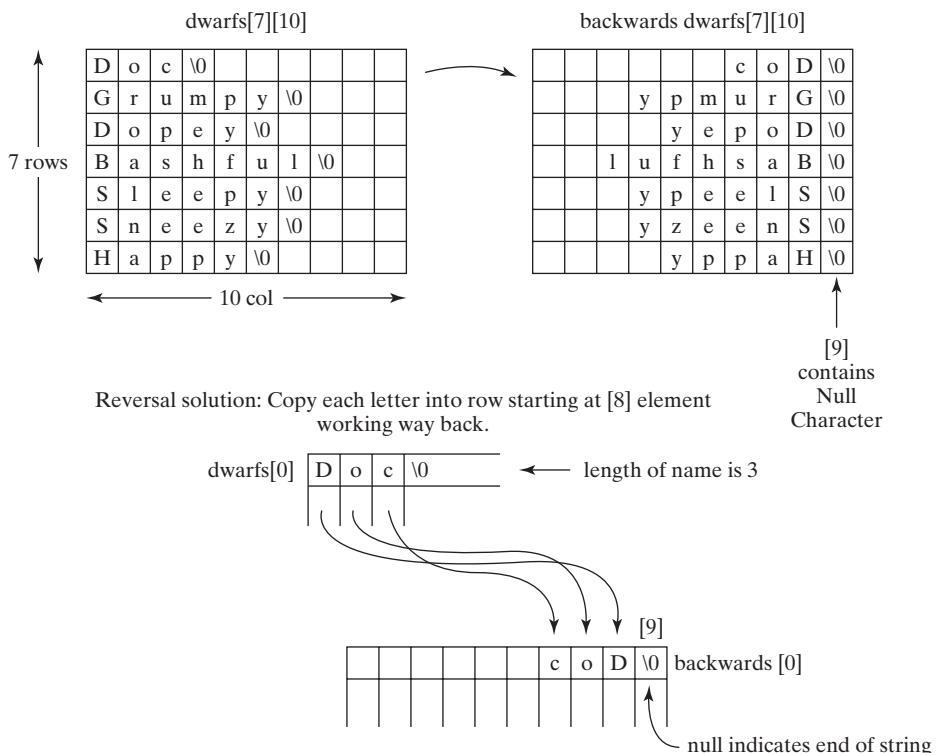


Figure 6-16
Reversed dwarfs.

Vertical Solution: Fill each column of vertical Dwarfs array with a row of dwarfs array.

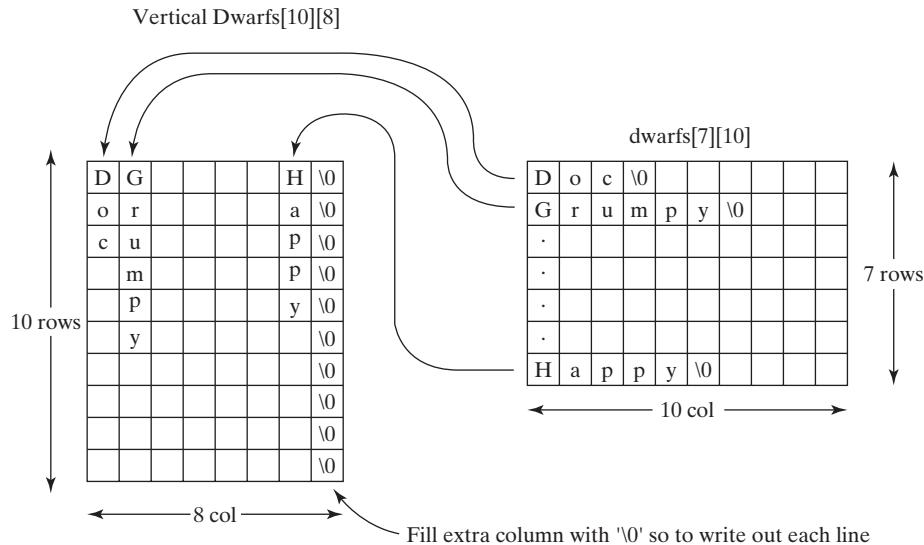


Figure 6-17
Vertical dwarfs.

Program 6-17

```

1 //File: FileSnowWhite.cpp
2 //A short program with the Seven Dwarfs
3 //Uses a 2-D Character String Array.
4
5 #include <iostream>
6 using namespace std;
7
8 #include "DwarfFunctions.h"
9
10
11 int main()
12 {
13     // Declare three 2-D arrays
14     char dwarfs[7][10];
15     char verticalDwarfs[10][8];
16     char backwardsDwarfs[7][10];
17
18     bool good = ReadNames(dwarfs);
19     if( !good)
20     {
21         cout << "\n Couldn't find the file. ";
22         exit(1);
23     }
24
25     TurnVertical(dwarfs,verticalDwarfs);

```

```

26     ReverseNames(dwarfs, backwardsDwarfs);
27     WriteArraysToScreen(dwarfs,
28                         verticalDwarfs, backwardsDwarfs);
29
30     return 0;
31 }
```

Now we'll examine the function prototypes contained in the header file. We pass the reference to the dwarfs array to all of the functions. The two functions that fill the vertical and reversed names receive the respective array references, too. The function that writes all three arrays to the screen have three array input parameters. Lastly, we place the `#define` statement for the input file in this header.

Program 6-17

```

1 //File: DwarfFunctions.h
2
3 bool ReadNames(char dwarfs[][10]);
4 void TurnVertical(char dwarfs[][10],
5                   char vertical[][8]);
6
7 void ReverseNames(char dwarfs[][10],
8                   char backwards[][10]);
9
10 void WriteArraysToScreen(char dwarfs[][10],
11                          char vertical[][8],
12                          char backwards[][10]);
```

The third source file contains the code for the dwarf functions as well as a `#define` statement that identifies `INPUT_FILE` as “`DwarfNames.txt`”. The comments explain the workings of the functions. In a nutshell, the reverse function takes each letter of each row of the dwarfs array and places it into a row of the backwards array, filling each row from the “right side” and working backwards. The vertical function takes the letters along a row and places it down the column of the vertical array. Refer again to Figures 6-16 and 6-17. Don't forget the importance of the null character! We need to be sure we have it at the end of each row so that the names are written correctly by `cout`.

Program 6-17

```

1 //File: DwarfFunctions.cpp
2
3 #include <iostream>
4 #include <fstream>
5 using namespace std;
6
7 #include "DwarfFunctions.h"
8 #define INPUT_FILE "DwarfNames.txt"
9
```

```
10 void WriteArraysToScreen(char dwarfs[][10],  
11     char vertical[][8], char backwards[][10])  
12 {  
13     int i;  
14  
15     cout << "\n Here are the 7 Dwarfs \n";  
16     for (i = 0; i < 7; ++i)  
17     {  
18         cout << dwarfs[i] << endl;  
19     }  
20  
21     cout << "\n Here are the 7 Dwarfs Backwards \n";  
22     for (i = 0; i < 7; ++i)  
23     {  
24         cout << backwards[i] << endl;  
25     }  
26  
27     cout << "\n Here are the 7 Dwarfs Vertical \n";  
28     for (i = 0; i < 10; ++i)  
29     {  
30         cout << vertical[i] << endl;  
31     }  
32 }  
33  
34 void TurnVertical(char dwarfs[][10],char verticalDwarfs[][8])  
35 {  
36     //initialize 0-6 col w/ ' ' and last with nulls  
37     for (int row = 0; row < 10; ++row)  
38     {  
39         for(int col = 0; col < 8; ++col)  
40         {  
41             if(col != 7)  
42                 verticalDwarfs[row][col] = ' ';  
43             else  
44                 verticalDwarfs[row][col] = '\0';  
45         }  
46     }  
47  
48     //Place first row into first column  
49     //fill letter by letter,  
50     for (row = 0; row < 7; ++row)  
51     {  
52         //obtain the length of each dwarf's name  
53         int length = strlen(dwarfs[row]);  
54  
55         //write dwarf's name down the column  
56         for(int col = 0; col < length; ++col)  
57         {
```

```
58         verticalDwarfs[col][row] = dwarfs[row][col];
59     }
60   }
61 }
62
63 void ReverseNames(char dwarfs[][10], char backwards[][10])
64 {
65   //Let's first fill all the elements with blanks
66   //and put a null in the last element.
67   for (int row = 0; row < 7; ++row)
68   {
69     for(int col = 0; col < 8; ++col)
70     {
71       backwards[row][col] = ' ';
72     }
73     backwards[row][9] = '\0';
74   }
75
76
77   //Now copy letters into each row
78   for ( row = 0; row < 7; ++row)
79   {
80     //obtain the length of each dwarf's name
81     int length = strlen(dwarfs[row]);
82
83     //write dwarf's name backward, fill from far end
84     //need 2 indexes, dwarf & backwards
85     int dwIndex = 0;
86
87     for(int col = 8; col >= 8-length+1; --col)
88     {
89       backwards[row][col] = dwarfs[row][dwIndex];
90       ++dwIndex;
91     }
92   }
93 }
94
95
96 //Function that reads the names.
97 bool ReadNames(char dwarfs[][10])
98 {
99   ifstream input;
100  input.open(INPUT_FILE);
101
102  //check that we've got the file open
103  if(! input)
104  {
```

```
105     cout << "\n ERROR Can't find " << INPUT_FILE;
106     return false;
107 }
108
109 //We'll read until the end of the file.
110 //Have to have an int for each array element.
111 int count = 0;
112
113 while(!input.eof() )
114 {
115     input.getline(dwarfs[count],10 );
116     ++count;
117 }
118
119 //all done reading, close the file
120 input.close();
121
122 return true;
123 }
```

Output

Here are the 7 Dwarfs

Doc
Grumpy
Dopey
Bashful
Sleepy
Sneezy
Happy

Here are the 7 Dwarfs Backwards

coD
ypmurG
yepoD
lufhsaB
ypeelS
yzeenS
yppaH

Here are the 7 Dwarfs Vertical

DGDBSSH
oroalna
cupseep
meheep
pyfpzy
y uyy
1

Colorado River Stream Gauge Flow Data

The water in the Colorado River begins its journey high in the mountains of Wyoming and Colorado, and meanders through the Grand Canyon, eventually supplying the population in southern California. There are power generation dams along the river as well. This water is of vital importance to the western United States. The U.S. Department of the Interior, Bureau of Reclamation monitors the natural flow and volume of water in the Colorado River. The U.S. government's website at <http://www.usbr.gov/lc/region/g4000/NaturalFlow/index.html> contains a wealth of information concerning this scientific research.

Government researchers have placed many stream gauges throughout the mountains of Wyoming and Colorado that measure the inflow values of the rivers and tributaries that feed into the Colorado. These gauges measure how much water is flowing through the streams. Scientists also try to estimate evaporation and local usage in order to gain an estimate of net water through the river.

For our last program, we'll practice using arrays by looking at one set of data for a stream gauge that is located on the Colorado River near Glenwood Springs, Colorado. The data for Stream Gauge 09072500 contains monthly flow rates from January, 1971, through December, 2003. Figure 6-18 shows the beginning and end of the comma separated data file obtained from the website listed above. The first piece of data is the month/year and the second is the flow measured in acre-feet per month. That is, for January, 1971, this gauge measured 71,642 acre-feet of water. (An acre-foot is the amount of water it takes to cover one acre of land in water one foot deep.)

Program 6-18 reads the data into an array then searches the array for the month that had the highest flow rate and the month that had the lowest flow rate. This data is written to the screen in the *main* function. Two things to study in this program: first, we are not exactly sure how many values are in the data

Month/Year	Flow (Acre-feet)
1/71, 71642	
2/71, 71531	
3/71, 104709	
4/71, 174102	
5/71, 348045	
6/71, 518646	
7/71, 271125	
8/71, 121109	
9/71, 109251	
10/71, 90364	
11/71, 85850	
12/71, 71860	
1/72, 67336	
2/72, 62286	
3/72, 94266	
10/02, 49735	
11/02, 51371	
12/02, 38616	
1/03, 38132	
2/03, 37427	
3/03, 50813	
4/03, 63030	
5/03, 226896	
6/03, 251187	
7/03, 90119	
8/03, 84239	
9/03, 83777	
10/03, 94692	
11/03, 48458	
12/03, 46438	

Figure 6-18

The StreamGaugeInflow.csv file contains data for 1/71 to 12/03. These images show the start and end of this data file.

file, so we'll size the array to be a bit larger than our estimate. (Better safe than sorry when working with arrays!) We'll read the data until it reaches the end of the file and count the values as we read them. Second, the function *FindHighLow* is passed the flow data, the total, and the references to two integers for the high and low indexes. Instead of needing separate data variables for the high and low values and the months in which they occurred, we'll use a simpler approach. We'll find the high and low values and return these two integer indexes. We can then use these indexes to report the volumes as well as the corresponding dates. The three source files are presented here in sequence, with comments explaining code details. The output is shown after the code containing main.

Program 6-18

```
1 //File: FlowDriver.cpp
2 //This program reads the flow data for a
3 //river gauge for the Colorado River.
4 //We find the high and low flows.
5 //We obtain indexes for high and low flow
6 //rate and use them to reference data arrays.
7
8 #include <iostream>
9 #include <string>
10 using namespace std;
11
12 #include "FlowFunctions.h"
13
14 int main()
15 {
16     cout << "\n This program finds the highest and"
17         << " lowest flow values \n for"
18         << " Inflow Gauge 09082500 in the "
19         << "\n Colorado River near Glenwood Springs, Colorado"
20         << "\n from 1/1971 to 12/2003. " << endl;
21
22     int flowData[400];           //33 years * 12 = ~400 months
23     string Dates[400];
24
25     int highIndex, lowIndex;
26
27     int total = ReadFlowFile(Dates, flowData );
28     if(total == -99)
29     {
30         cout << "\n Trouble reading input file.";
31         cout << "\n Exiting program. " << endl;
32         exit(1);
```

```

33      }
34
35      FindHighLow(flowData, total, highIndex, lowIndex);
36
37      cout << "\n The highest flow was " << flowData[highIndex]
38          << " acre-feet per month in " << Dates[highIndex] << endl;
39
40      cout << "\n The lowest flow was " << flowData[lowIndex]
41          << " acre-feet per month in " << Dates[lowIndex] << endl;
42
43      return 0;
44  }

```

Output

This program finds the highest and lowest flow values for Inflow Gauge 09082500 in the Colorado River near Glenwood Springs, Colorado from 1/1971 to 12/2003.

Reading StreamGaugeInflow.csv

Read 396 flow values.

The highest flow was 792845 acre-feet per month in 6/84

The lowest flow was 36710 acre-feet per month in 12/01

Program 6-18

```

1  //File: FlowFunctions.h
2
3  #define FILE "StreamGaugeInflow.csv"
4
5  #include <string>
6  using namespace std;
7
8  //Read flow data from FILE file.
9  //Returns the total lines read from file.
10 //If error, returns -99
11 int ReadFlowFile(string Dates[], int flowData[]);
12
13 //Search data for the high and low flows.
14 //Return the indexes.
15 void FindHighLow(int flowData[], int total,
16                  int &rHigh, int &rLow);

```

Program 6-18

```

1  //File: FlowFunctions.cpp
2

```

```
3 #include <iostream>
4 #include <string>
5 #include <fstream>           //for ifstream, ofstream
6 using namespace std;
7
8 #include "FlowFunctions.h"
9
10 //Search data for the high and low flows.
11 //Return the indexes for later use.
12 void FindHighLow(int flowData[], int total,
13                   int &rHigh, int &rLow)
14 {
15     //set high and low indexes to first value
16     rHigh = rLow = 0;
17
18     //local vars to keep track of high/low values
19     int highFlow, lowFlow;
20
21     //set to first value
22     highFlow = lowFlow = flowData[0];
23
24     //now traverse array looking for high/low
25     for(int i = 1; i < total; ++i)
26     {
27         if(flowData[i] < lowFlow)
28         {
29             lowFlow = flowData[i];
30             rLow = i;
31         }
32
33         if(flowData[i] > highFlow)
34         {
35             highFlow = flowData[i];
36             rHigh = i;
37         }
38     }
39 }
40
41 int ReadFlowFile(string Dates[], int flowData[])
42 {
43     ifstream inFlow;
44     inFlow.open(FILE);
45
46     //check that we've got the file open
47     if(! inFlow)
48     {
49         cout << "\n ERROR Can't find " << FILE;
50         return -99;
51     }
```



```
52
53     cout << "\n Reading " << FILE << endl;
54
55     //use i to count entries in file
56     int i = 0;
57
58     //Here is a sample line:
59     //1/71,71642
60     //The comma separates date and flow
61     while(!inFlow.eof() )
62     {
63         //read date using ',' for delimiter(stop)
64         getline(inFlow, Dates[i], ',');
65
66         //getline has removed the comma,
67         //now just read the flow number
68         //use cin-like form to read numbers
69         inFlow >> flowData[i];
70
71         //get rid of \n at end of line
72         inFlow.ignore();
73
74         //increment the count
75         ++i;
76     }
77
78     cout << "\n Read " << i << " flow values. " << endl;
79
80     //all done reading, close the file
81     inFlow.close();
82
83     //return total number of values
84     return i;
85 }
86
```

■ REVIEW QUESTIONS AND PROBLEMS

Short Answer

1. What data type must be used as an array index?
2. What is meant by the term *zero-indexed*?
3. What type of array is a C-string?
4. Name three situations in which data are maintained ideally as an array.
5. Is it possible to have an array of C-strings? If so, give an example of the declaration.

6. What does the size value represent in an array declaration statement?
7. Why is the null character important in C-strings?
8. Can you null-terminate a numeric array?
9. When passing an array to a function, what is actually passed to the function?
10. What technique can a programmer use when he is trying to find the minimum or maximum value in an array?
11. Can the *BubbleSort* be changed so it sorts from high to low? What do you need to change in the code to make this happen?
12. Why is the *for* loop a handy tool when working with arrays?
13. If you declare two arrays sized to 10 and you attempt to assign values to both arrays out to the 12th element, what is the result of this action?
14. What do you see if you *cout* the name of an array?
15. Is there a way to write the entire contents of a numeric array in one *cout* statement?
16. What is meant by the term “going out of bounds” on an array? Does C++ automatically check and stop you (the programmer) from going out of bounds with your array? Explain.
17. List a set of data that could be stored in a one-dimensional array. List data for a two-dimensional and three-dimensional array too.
18. When you create an array and pass it to a function, a pointer is actually passed to the function. Explain how this is the C language’s original call-by-reference technique.
19. What happens if you forget to place a null character in a character array and then write the array using *cout*?
20. Is it possible to make an integer array bigger once the program begins to execute? Explain.

Debugging Problems: Compiler Errors

Identify the compiler errors in Problems 21 to 24 and state what is wrong with the code.

21.

```
int list{25},i;
float a, b, c;
for(i = 0, i<25; ++i)
{
    listi = 0.0;
}
```

22.

```
#include <iostream>
using namespace std;
```

```

int main()
{
    float numbers[100];
    int j;
    cout << numbers;
}

23.
#include <iostream>
using namespace std;
char [] FillArray();
int main()
{
    int values[75];
    values = FillArray();
    return 0;
}

24.
#include <iostream>
using namespace std;
void SortArray(int values[]);
int main()
{
    int values[75];
    //assume values becomes filled with data
    SortArray(values[75]);
    return 0;
}
void SortArray(int values)
{
    for(i = 0; i < 75; ++i)
    {
        if(values[i] < values[i-1]);
        values[i] = values[i-1];
    }
}

```

Debugging Problems: Run-Time Errors

Each of the programs in Problems 25 to 28 compiles but does not do what the specification states. What is the incorrect action and why does it occur?

- 25.** Specification: Fill a 100-element floating point array with the values 0.01, 0.02, ..., 0.99, 1.0.

```

#include <iostream>
using namespace std;
int main()
{
    float x[100];

```

```

int i;
for(i = 1; i <= 100; ++i)
{
    x[i] = i/100;
}
return 0;
}

```

- 26.** Specification: Ask the user for a character array and then reverse the characters. If the user entered Hello World, the new string would read *d1row o11eH*.

```

#include <iostream>
using namespace std;
int main()
{
    char saying[50], revSaying[50];
    int i;
    cout << "\nEnter a saying.";
    cin.getline(saying, 50);
    for(i = 0; i < 50; ++i)
    {
        revSaying[i] = saying[50-i];
    }
    return 0;
}

```

- 27.** Specification: Fill a character array with the uppercase alphabet (place A in [0], B in [1], etc.).

```

#include <iostream>
using namespace std;
int main()
{
    char alphabet[26];
    int i;
    for(i = 0; i < 26; ++i)
    {
        alphabet[i] = i;
    }
    return 0;
}

```

- 28.** Specification: Assume that two arrays, *x* and *y*, are both fifty-element arrays and that they contain double values. This function is supposed to switch the lowest *x* value with the lowest *y* value; that is, the lowest value in *x* is switched with the lowest value in *y*.

```

void Swticher(double x[], double y[])
{
    double low_x = 0, low_y = 0;
    int i, i_x, i_y;

```

```

for(i = 0; i < 50; ++ i)
{
    if(x[i] < low_x)
        low_x = x[i];      //find low x
    i_x = i;   //remember x index
    if(y[i] < low_y)
        low_y = y[i];      //find low y
    i_y = i;   //remember y index
}
y[i_y] = low_x;
x[i_x] = low_y;
}

```

Programming Problems

Write complete C++ programs for the following problems. Many of these problems read or write data files. Refer to Appendix F, “File Input/Output” for reference. Make sure that any data files you create to read data into your program are built in Notepad and do not have extra blank lines at the bottom of the file!

29. Write a complete C++ program that asks the user to enter text into a character array (a C-string). Send the text to a function called *ReverseIt*. This function will fill a second C-string so that the original string is reversed (as described in Problem 26). Limit the size of the C-strings to fifty characters. The last character in the original string (before the null) should be the first character of the second string. Write both C-strings from main. Incorporate a loop so that the user can continue to enter strings until he chooses to stop.
30. Create a data file with twenty-five values (one value per line) that represent the length of an oak leaf. (Hint: can you write a program to do this for you?) Have the values range from 1.000 to 6.000 (here 3 decimal places) inches in the x.xxx format. Write a program that declares an array that will hold these twenty-five values and pass the array to a *ReadFile* function. The *ReadFile* function asks the user for the name of the leaf file. It returns a Boolean indicating file open/read status. That is, it returns a false if the program was unable to open the file, a true if the file was opened and successfully read, filling the array. Pass the array to a *FindAve* function which returns the average value. Next, sort the array using a modified *Sort* function from Program 6-6, page 309. Send the array and average to the *Write* function that writes the average values, the three largest and three smallest leaves.
31. Create a data file with a first line that specifies MALE or FEMALE and a second line that tells how many data values are given (it will be a value between 10 and 20). Following this number are ten to twenty rows of data. Each row has two values, height (inches) and weight (pounds). For example, the file has data for eighteen females:

FEMALE

18

63 115

```
72 165
:
:
60 133
```

Your program should read this data file (read until EOF) and place the values into a two-dimensional array where the first column represents height and the second column represents weight. Next, call a function named *TallestAndLightest* that returns the tallest and lightest values, respectively. Print all the data to the screen.

32. Write a program that declares two twenty-element, one-dimensional integer arrays. Your program should fill these arrays with random numbers by calling a function named *FillArray*. You will call *FillArray* twice, once for each array. Use a separate function to obtain the a seed value as well as the integer range (such as 1000 to 2000, or -5 to 5) for each array. (You will call this twice, too.) The *Ask* function should check that the low range value is less than the high range value. Ask the user to re-enter the values if the ranges are not correct. Once you have this data, make two calls to *FillArray*. This function is passed the array to be filled, the size of the array, the seed value, and low and high range values. *FillArray* passes the seed to *srand()* and then fills the arrays with values between the low and high range. Once both arrays are filled, pass the arrays and seed values to the *WriteFile* function. This function asks the user for the name of the output file. Write the two arrays in a tabular format to this file, showing the seed values, the random range, and the twenty-five array element values.
33. This program is going to do a crude analysis of the *rand()* function we use for generating random numbers. We'll generate three sets of random numbers and plot them in Microsoft Excel to see how "random" our results appear. In main, declare three integer arrays, one sized to 20, one sized to 200, and one sized to 2000. Call the *FillArray* function (described in Problem 32) three times, *resetting the seed to the same value* each time you call *FillArray*. The random numbers should have the range between and including -50 to +50. Call a *WriteFile* function that is passed the three arrays. The *Write* function obtains the root of the filename from the user. Do not have the user enter an extension! You will build the three filenames from this root filename. Suppose your user enters "RandData" as the root name. The three files you write will be *RandData_20.txt*, *RandData_200.txt* and *RandData_2000.txt*. Each file contains one element per line. Using Microsoft Excel, open each output file and create a scatter plot of the data. How random does your data appear? What do you expect to see?
34. Let's play Bingo! In this chapter we developed the code necessary to make a Bingo card. (See Program 6-11, page 325 and Program 6-12, page 329.) In our *main* function we'll set up one Bingo card and call the *FillCard* function. Next, we'll need to begin a loop that calls the *GetNumber* function—which generates a random number simulating the caller drawing a number in the Bingo game. Read Program 6-11, 6-12 descriptions to learn the values that this function generates. Use a static array in *GetNumber* to keep track of which numbers have been used. Do not return the same number in a Bingo

game. Include a reset flag that can be passed into *GetNumber* to clear the used number array in preparation of a new game.

You'll need to have a second two-dimensional array that keeps track of what squares have been marked. This array may be either integers or characters. Write a function called *MarkCard*, which is passed the card array, this marked array and the current (called) Bingo number. It checks to see if the called value is on the card.

In order to watch your progress, pass your Bingo card and this second marked array to a new *DisplayCard* function. This function draws the card as before, but displays an "0" or "*" in the card position where a marked number is located. Note, the data in your Bingo card is not altered. The marked grid keeps the status of your game. You'll have to use both arrays to be able to accurately display the card data.

Last, you'll need to check your marked card to see if you have a Bingo. In this program, we'll call the *CheckForXBingo* function, which returns true/false concerning your Bingo status. It checks for both diagonals marked, as in an "X" pattern. Have your *play* loop continue until you obtain a Bingo. With each number drawn mark your card, display your card, and check it for a Bingo. You should give your user an easy way to watch and exit out of the play loop. Your display information can report also the number drawn and total numbers drawn.

35. As you know, most monthly calendars are usually a grid of six rows by seven columns, filled with numbers representing the month's days. The day of the week is above each column, beginning with Sun, ending with Sat. Across the top of the grid is the name of the month and year.

In this program we'll write a C++ program that creates, fills, and writes a file showing a calendar month with its appropriate dates. In *main* declare a *monthGrid* integer array sized six rows by seven columns and a string for the month name. Fill a string array with the names of the week, beginning with "Sun" and ending with "Sat". Fill a string array with the names of the months, along with an integer array holding the total days for each of the twelve months. That is, the *monthName[0]* is January, and *monthDays[0]* is 31.

In the *AskForMonth* function, the user selects the month to be displayed. Use the accepted month-integers for obtaining this, i.e., January is 1, February is 2, etc. Also ask the user to enter what day of the week is the first day of the month. Ask for the year too. The *FillMonthGrid* function fills the dates into their appropriate grid position. This function contains the necessary code to determine if February is in a leap year. Consult Chapter 3, Problem 39 (page 171) for assistance. The *WriteMonth* writes a MonthNameYear.txt output file, such as September2007.txt. In the file is a neatly laid out month including the name centered across the top of the page, the day names across the top of the columns and the individual dates spaced along the rows. Obviously, we can't perform special formatting and font or graphics with this file. If the month grid is neatly displayed, that is the ultimate goal. You may consult the Appendix D, "ASCII Character Codes" to see the graphical characters available to you. Don't spend a lot of time trying to do some fancy graphics—concentrate on learning to work with two-dimensional arrays!

Extra challenge! There are algorithms available to that will calculate the day of the week. Are you game to find one of these algorithms and incorporate it in this program so that your *WriteMonth* function prints an accurate calendar? Your user will enter the desired year and month and your program does the rest of the work. Good luck and happy hunting!

36. For comparison's sake, let's rewrite the *ReverseIt* problem presented in Problem 29 using C++ string objects instead of character arrays, i.e., C-strings. Ask the user to enter some text and read it into a string object. Send the text object to the *ReverseIt* function. This function will fill and return a string object with the reversed text of the original string. Write both the original and reversed strings from *main*. Hint: the *stringstream* object will help. Refer to Chapter 4, Problem 29 (page 239) for help. If you wrote both programs, can you tell what part of this program is easier with the C-string? What part is easier with the string object?
37. Create a data file filled with between 250 and 300 integer values. These values should have a range of 0–100 (inclusive). (See Problem 32 and 33 for help with this.) Now write a program that has an integer array declared in *main*. It should be sized to 300. We'll read our data file in the *ReadFile* function. It asks the user to enter the name of this data file and fills the array with these values. It returns the total number of integers read as well. The *main* should then call the *Sort* function, which sorts the values low to high. The program has a *do while* loop that asks the user to enter two integer values using the *Get2Integers* function. The *SearchIt* function is then called. It is passed the integer array of values, the two numbers, and it searches the array for these numbers. The *SearchIt* function sets flags true or false indicating whether or not it found the numbers (use boolean flags). The *main* function should report the results of the search, stating if it found neither, one (which one), or both.
38. You are to write a program that models Powerball¹ (the lottery game) using one-dimensional arrays. (See www.powerball.com for complete game information.) In Powerball there is one red ball (the powerball) that can be a value between 1–42. There are five white balls and their values can be between 1–55. For one dollar, you pick five white balls values (five different numbers between 1–55) and one powerball. Then, twice a week, the game draws one red ball and five white balls. If your white balls and powerball numbers match the game's white balls and powerball—you win millions of dollars!!!

In this game, ask the user to enter his name. Use a function (you name it) to ask the user for his powerball input (i.e., the five whites and one red ball). Note: the user should be able to enter the five white balls in any order. Then call a different function that generates the computer's powerball white balls and red ball. Your program should write the user's entry and computer's entry to the screen. (For an extra challenge tell the user the number of white balls

¹Powerball® is a registered trademark of the Multi-State Lottery Association. (MUSL) is a non-profit, government-benefit association owned and operated by its 31 member lotteries. Each MUSL member offers one or more of the games administered by MUSL. All profits are retained by the state lottery and are used to fund projects approved by the state legislatures.

that match the game balls, and if the powerballs match!) Ask if he would like to keep playing. For each game the user enters his numbers and the computer determines a new set of numbers. You should use the user's name when writing anything to the screen.

For both sets of white balls, ensure that each white ball is unique. If the user tries to enter the same number twice, the program says that number has been used. Ask the user to enter a different value. If the random number generator gives the same white ball, you must ignore it and keep calling *rand()* until all five balls are unique. Hint: there are two different ways to ensure unique numbers. You can use a check-array like we do in the Bingo problem or you can sort the array and make sure adjacent values are not the same.

39. In this program we are going to write a C++ program to help us analyze the Powerball Game twice-weekly “draws” to determine the randomness of the results. Read Problem 38 and visit www.powerball.com for information on this game. The goal of this program is to create an output file that shows the value of the game ball, the actual number of times it has been drawn, the expected number of times it should have been drawn (if truly random) and the difference between these values.

For example, there are 42 possible red, Powerball values, ranging from 1–42. If we analyzed 420 draws, we'd expect to see each Powerball (red) a total of 10 times ($420/42$). Each game draws five white balls, ranging from 1–55. If we analyzed 100 draws, then we drew the white balls 500 times. We'd expect to see each white ball 9.09 times ($500/55$). Your program will analyze the data from August 31, 2005 to the current draw date and determine the randomness of the draws.

You need to visit the Powerball site and follow the links to Powerball numbers and Historical Numbers so that you can obtain the winning numbers in plain-text format. Cut and paste the winning numbers into a *.txt file, from the current date, back to 8/31/2005 when the range of white balls grew to 1–55. Be sure there are no blank lines at the bottom of your file! (Use Notepad.)

Once you have created your data file, you will write your program using two functions. In *main*, create two integer arrays, one sized 43 (elements 0–42) for the red, Powerball, and one sized 56 (0–55) for the white balls. (We'll not be using the 0th array element.) Your *ReadFunction* will open the data file and read each line of data, counting the white balls and red balls. Our two arrays are used to hold the count for the ball value occurrence, i.e., *red[1]* hold the number of times the Powerball of value 1 occurred, *red[2]* counts the 2's, etc.

After the data file is read and arrays are filled, pass them to the *WriteResults* function. Here it asks the user for the output file name, and write the results in a tabular format. The output file should look something like this:

Input File Name		Total number of Powerball Game draws in this file:		
Powerball Results				
Powerball number		Expected number	Actual	Difference
	1			
	2			

```
3
:
:
42
```

White Ball Results

White ball number	Expected number	Actual	Difference
1			
2			
:			
:			
55			

Note: one challenge in this program is reading the data from the plain-text file. Suppose these are the three latest draw results for December, 2006:

Draw Date	WB1	WB2	WB3	WB4	WB5	PB	PP
12/09/2006	52	22	19	34	30	40	4
12/06/2006	38	36	35	07	34	14	4
12/02/2006	42	34	50	45	17	11	3

The PP represents the power play value, which we'll ignore. We don't need the date value either. One way to read this file is to read the entire line into a C-string and assign the numeric values into a temporary character array. Copy the numeric values into a temporary character array and use the *atoi* function to convert to an integer. Here's a start for you. We read the line containing the column headings before we enter the processing loop:

```
char line[50],buf[5];
input.getline(line,50); //reads line column headings
//assume white is the white ball count array
int ball_marker, w;
while(!input.eof()) //input until end of file
{
    input.getline(line,50); //reads line of data
    ball_marker = 12; //sets index to first ball
    //for loop to read in 5 white balls
    for (int i = 0; i < 5; ++i)
    {
        buf[0] = line[ball_marker];
        buf[1] = line[ball_marker+1];
        buf[2] = '\0';
        //convert to int, the WB values is now an int
        w = atoi(buf);
        //now count the value
        white[w] += 1;
        //advance index to next ball
    }
}
```

```

    //Now get the PB and count it in the red array.
    //I'll let you figure out the last part of this.
}

```

40. Program 6-18 (page 352) presented a program that used data for a stream gauge that is located on the Colorado River near Glenwood Springs, Colorado. The data for Stream Gauge 09072500 contains monthly flow rates from January, 1971, through December, 2003. Below shows the beginning of the comma separated data file obtained from the text's website. The first piece of data is the month/year and the second is the flow measured in acre-feet per month. That is, for January, 1971, this gauge measured 71,642 acre-feet of water. (An acre-foot is the amount of water it takes to cover one acre of land in water one foot deep.)

```

1/71,71642
2/71,71531
3/71,104709
4/71,174102
5/71,348045
6/71,518646

```

In this program we'll write a C++ program that reads the data into an integer array (flow data) and string array for the month/year. Size the arrays to 400. This program should then determine several statistical values for this data. First, find the average value for the flow data and the median value for the entire data set. This is a good exercise, but not very informative because the flow data is cyclic. That is, as the snow melts and the volume in the stream will be high. This occurs in the May and June. In the winter, there is less volume due to the freezing temperatures, snow, and ice pack. Therefore, this program should also determine the peak flow average and median (using the May months) and the lowest flow data (using February months).

To review, the median value is the value that has half the values above it and half the values below it. For example, suppose we had these seven values:

39, 73, 72, 27, 33, 40, 48

the best way to find the median would be to sort the values, low to high

27, 33, 39, 40, 48, 72, 73

The median would be 40 because there are 3 values below 40 and three values above 40. (If there are an even number of values, you find the average of the two middle values.

Your program should declare your arrays in *main* and then call a *ReadFile* function, which fills the two arrays. It should read until the end of file and keep track of the total values. You should write two functions, *FindMedian* and *FindAve*. Each function is passed an integer array and total in the array, and it finds the median and average values, respectively. For you to determine the peak and low flow data, write *FindPeakAveAndMedian* and *FindLowAveAndMedian* functions. In these two functions create arrays filled

with the pertinent months' data (May for peak, February for low flow). These functions should each call the *FindMedian* and *FindAve* functions.

Your program call the *WriteResults* function. It is passed all the relevant data, include time period, gauge name, location, and average and median values for the group, peak and low flow. The data should be written to an output file. Ask the user to enter the name of the output file. This output file should be neatly formatted, as if it were going to be included in an official, final report.

41. You are to locate data on the web and write a program that performs analysis on that data. There is a wealth of data available to you. My students found crime data from the FBI, weather data from noaa.gov, population data, airline on-time data, drunk driving statistics, water use data from USGS.gov, just to name a few.

Description document: You need to write a description of the data, what it represents, exactly where you obtained it (give website), and explain the three pieces of information you will determine from the data. Give a sample of the data file. This should fully describe your data and what you plan to do with it.

Your program should have a Read function which reads the data file. Your program may use either arrays or vectors to store data. You must use a file that contains at least 25 data points. (That is, if you are doing water sample, the data must have at least 25 samples.) There should be an Analyze function that crunches the numbers and finds your results. Note: if date data is in your file, and you are finding the “highest” data item, you should tell what date it occurred too. The *Write* function is passes all relevant data and asks the user to enter a filename for the output file. The output file should have your name, a description of the data, location, and state results of your research. By reading the description document (above) and the output file, the reader should know your data and your results.

Note: when reading your data file, you may need to “read past” lines that are comments or table headers. You may find the ifstream’s *peek()* function helpful. It returns the first character of the next line to be read (doesn’t remove it). You can peek at the first character to help you decide what to do with the next line. Here’s a short code sample:

```
string skip;
while( !input.eof())
{
    char c = input.peek();
    if( c == '#')           // read past lines beginning with #
    {
        getline(input, skip);
        continue;
    }
    //read line, obtain relevant data
    :
}
```



7

Classes and Objects

KEY TERMS AND CONCEPTS

access specifiers
array of objects
class
class constructor
class declaration
class definition
class destructor function
class members
instance of a class
method
object
obtaining computer system time
operator function
overloaded constructors
overloaded operators
private
public

KEYWORDS AND OPERATORS

class
operator
private
public
scoping operator ::

CHAPTER OBJECTIVES

Introduce the principles and definitions used in object-oriented programming.
Illustrate how classes incorporate data and functions.
Explain the relationship between the class declaration and objects.
Demonstrate how a class declaration is written.
Show how the “world” may access public class members.
Present several simple program examples that illustrate the mechanics of working with objects.
Describe the purpose of class constructor and destructor functions, and illustrate how these functions are used in a program.
Illustrate how pointers and references to objects work.



Who's Job Is It?

H ave you ever worked for a company where everyone had a specific job title and responsibilities? If you were employed in the airline industry, you know the various jobs include pilots, flight crew, mechanics, baggage handlers, and reservation-staff, just to name a few. Perhaps you've spent time working in a restaurant where the various job titles include managers, cooks, wait-staff, host or hostess, bus-boys and girls. Now, if you were a waiter, what did you do? You took customers' orders, picked up, and served their food. You didn't cook the food, nor did you order food from the supplier. If you were a baggage handler for an airline, you didn't fly the plane nor help customers make reservations.

Are you asking yourself why we are off track talking about restaurants and airlines when we should be learning how to write our own classes? Before I answer that, let's suppose you want to start your own business, say a bike messenger service in the downtown area of a large city. You have to identify the various jobs at your shop. What types of workers do you need? (Office manager, scheduler, messengers?) Then for each job, you need to identify the knowledge required to perform that job and its assigned tasks. As a programmer, writing object-oriented programs, you have to examine all of the program goals and identify the type of "workers" you need. Asking yourself what classes you need and what each class is supposed to do is an important part of designing your program. This is the same type of exercise that the new business owner must do when identifying jobs and tasks. You decide what tasks are appropriate for your new classes and what things each class will "need to know" in order to execute specific tasks. Don't forget, there may be C++ classes available for you to use too. Isn't that nice?

"Who's job is it?" is an important question when dealing with classes and objects. Keep that thought in mind as we move along here, and it will serve you well as you design your own classes.

7.1

What Do We Know About Classes and Objects?

Let's review what we know about C++ classes before we jump into writing our own. What classes have we used so far? Here is a short outline:

string	for handling text data
vector	for lists or groups of data, such as a list of strings or numbers
stringstream	for creating a formatted string
queue	models a line, (first in, first out) such as a bank line or a play list for songs
ifstream	open and read data from a data file
ofstream	open and write data to a data file

For each of these classes, we include the appropriate library at the top of the file. Inside a function we make an object that works for us. To use a class function, we ask the object to perform a task by writing the object and a dot operator to access the class' function. The class is the job description for the object—it contains the task list for the object.

The string class is handy for text data. We assign text into a string. The string class' "job description" gives us (among other things) the ability to assign, determine length, and find a substring in the string.

```
string sText; //make a string object named sText
sText = "C++ classes are fun!"; //use = to assign
int length = sText.size(); //size() gives us length
int where = sText.find("fun"); //find() searches for us
```

We used the vector class in Chapter 3's Program 3-18 when we searched for a woman's name. We stored the name and origin data in two separate vectors. To use a vector object, we tell the object what type of data it will contain. We add items to the vector using the *push_back()* function. We ask the vector how many items are in the list with the *size()* function. The *at()* returns the item given an integer index. Here are a few lines of code using a vector object:

```
vector <string> vNames; //make a vector object, vNames
vNames.push_back("Mary"); //load names using push_back()
vNames.push_back("Carol");
vNames.push_back("Elizabeth");
int number = vNames.size(); //size() tells us how many names are
                           stored
cout << vNames.at(0); //at() returns the first name
```

Our C++ Bank program in Chapter 5 (Program 5-8) incorporated a queue object. Do you remember how to make and use a queue? We include the library, make an object, then have the object perform tasks for us. The task list is defined in the class. (Same song, different verse, right?) Here we make a *bankLine* queue object

for customers (string objects), put three in the line, then serve and remove the first one.

```
queue <string> bankLine;           //make a queue object bankLine
bankLine.push("Richard");          //add to the end of the line
                                   using push()
bankLine.push("Roberto");
bankLine.push("Kristy");
cout << "Serving " << bankLine.front(); //front() returns the first name
bankLine.pop();                   //remove the first element from
                                   the line
```

If you haven't discovered Appendix G, "Partial C++ Class Reference," now is the perfect time for you to look at this appendix. Here you can scan through a partial listing of the string, vector, and queue class functions—in essence, these functions show you their job descriptions. The designers of these C++ classes thought about the class tasks when they wrote the class specifications. We'll be doing the same in this chapter.

7.2

Writing Our Own Classes

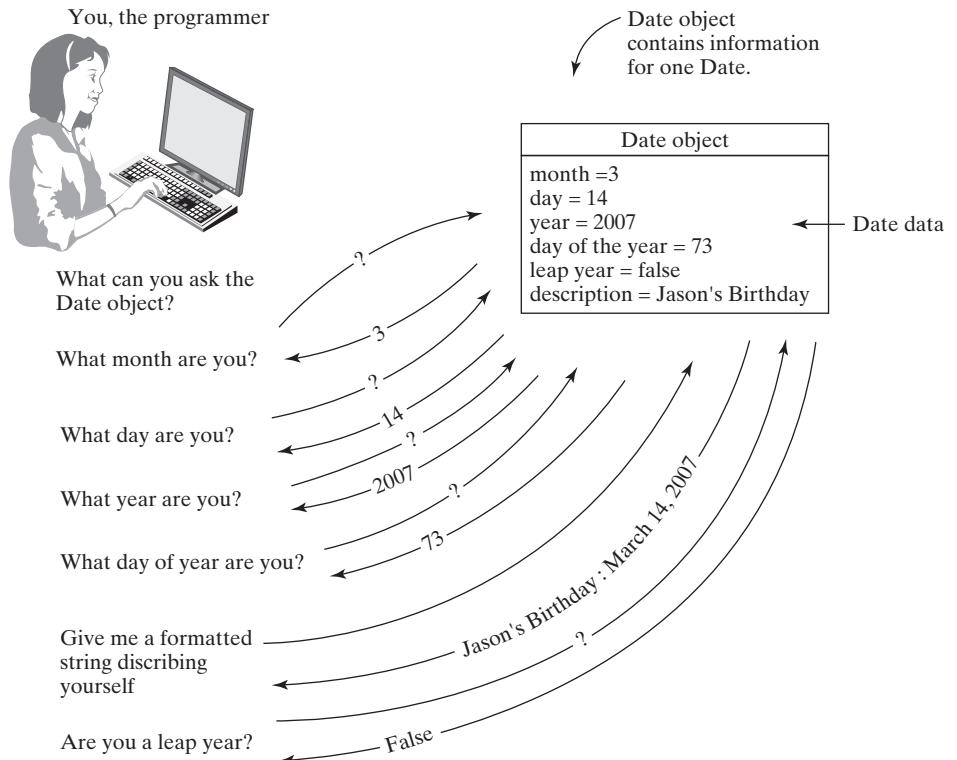
Hand-Waving Example: Our Date Class

No time like the present to design our own class. Let's talk through a simple example instead of getting bogged down with object-oriented principles, terminology, and C++ syntax. The time to worry about semi-colons and braces will come soon enough. For now, let's pick an easy class and walk through its design. In this discussion, we'll layout the features for a Date class. After we design it, we'll write it in C++ code, and see how it works. It will then be ready for us to use throughout these last two chapters.

What is the date today? If you're like most of us, you'll have to stop a minute and think—unless it is an important date like your birthday! Now, did you think of today's date in terms of a month name, day, and year, or as three integers? The exact form doesn't matter—what does matter is that you know that the date data is represented using three integers, or two integers and a string. A date can be described using many formats. Here are a few ways to write the same date:

March, 14, 2007	Mar. 14, 2007	14-Mar-2007	14/3/2007
3/14/2007	03/14/07	14-March-2007	14/3/07

Regardless of the format, we have four pieces of data with the date, i.e., the month, day, year (integers), and month name (string). What other information might you need when talking about a date? You may need to know if your date is a leap year. The year 2007 is not a leap year. A useful date value to know is what day of the year it is. January 1 is the first day of the year. February 1 is the 32nd day of the year. Non-leap years contain 365 days. Leap years contain an extra day: February, the 29th. Therefore, in a non-leap year March 14 is the 73rd day of the year (obtained by adding 31 + 28 + 14). One last feature of our customized Date class is that it will have a

**Figure 7-1**

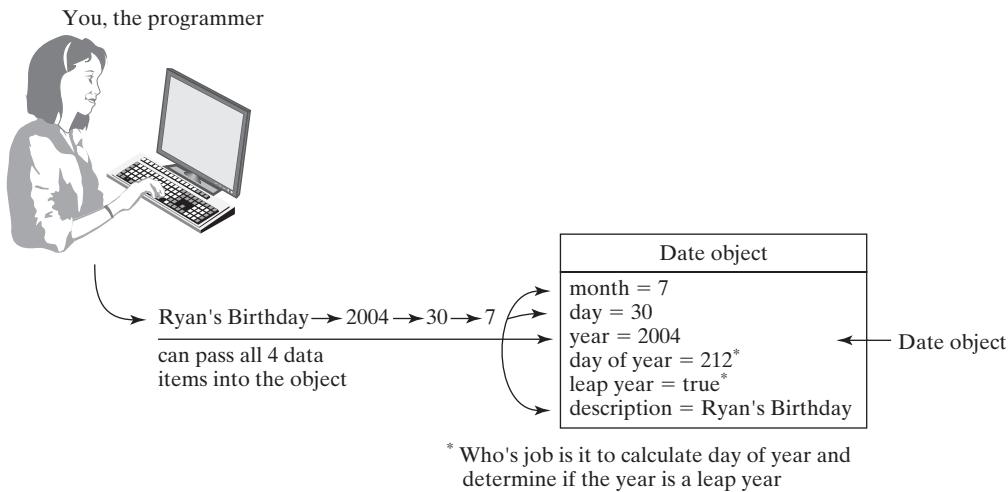
Information available from our date object.

string variable to hold a description so that it “knows” what the date represents—such as “Jason’s Birthday”, “first day of school”, or “driver’s license expiration”.

When you write a class, you must think about the type of tasks (jobs) the class objects will do for you. Our Date class holds date information for us, so what would you like your Date object to do for you? Obviously, our Date object tell us the month, day, year, and day of the year. It would be nice if it gave us a formatted string of itself—such as “Jason’s Birthday: March 14, 2007”, as well as be able to tell us if it is a leap year. Look at Figure 7-1 and see all the questions that our programmer can ask our Date object.

If you look at our Date object in Figure 7-1, it contains data for Jason’s Birthday. How did the data get into the Date object? There are a few ways to set the Date object’s data. Let’s look at one way here. We need to be able to tell the Date object what date it is holding and its description. Let’s allow our programmer to pass three integers and one string into the Date object. This means we can change the date information in our object anytime we wish. Let’s set our Date object data to 7/30/2004, with “Ryan’s Birthday”. Figure 7-2 shows this. (Wait! Who’s job is it to determine the day of the year? What about that pesky leap year?)

As class designers we must think about what happens when our object is constructed. We have to be sure that the class data in the object are initialized to

**Figure 7-2**

One way to set the date data into the date object.

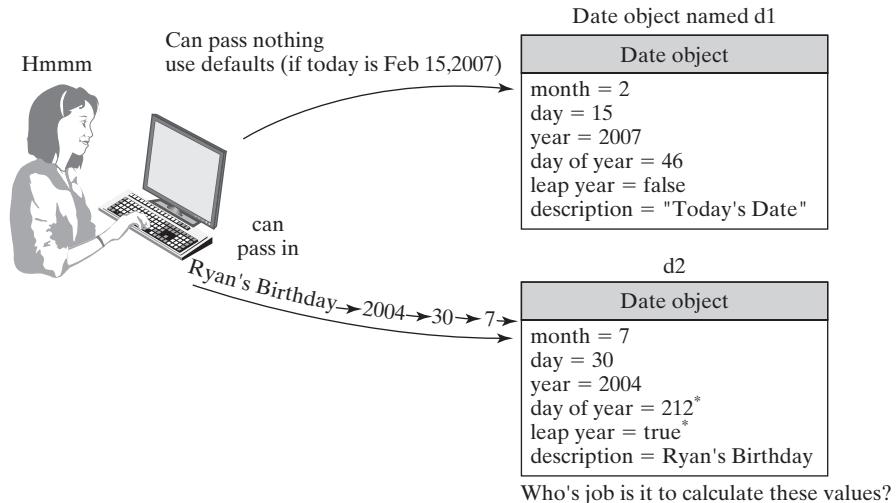
appropriate values. (We do not want our object to contain “trash” values, do we?) If we don’t tell the object what its data is when we create it, we must have the class assign values to the data. These are the “default” class data values. What do you think an appropriate “default” date should be for our Date object? Initially, we can assign anything into our Date object, such as January 1, 2000, or mid-year, June 30, 2007, but that is not practical. Let’s design our Date class so that it is assigned today’s date along with the description of “Today’s Date”. We can obtain the date from the computer’s operating system because C++ gives us tools to do just that.

Programmers like having a variety of options when working with classes. We will write the Date class so that the programmer can pass in his own data when the object is created, instead of being stuck with the default data. After all, it is reasonable to expect that a program might need several Date objects, all containing different date data.

For our Date class, you can create Date objects in two ways. First, by default, the system date and “Today’s Date” is assigned to the object’s variables. Second, the month, day, year, and description are passed to the Date object when it is created. By giving the programmer two options, this makes our Date class convenient to use. See Figure 7-3.

Our First C++ Class: The Date Class

C++ provides the programmer with the necessary tools to build his own classes. We’ll use the multi-file source code form for our class code as it is a tidy way to organize our programs. In this section, we’ll run through the source code for our Date class so that it is written to match the way we designed it in the previous section. We’ll touch on all the parts and pieces with brief explanations—and then revisit each part in more detail. Don’t feel overwhelmed as you read through this part of the chapter. Everybody needs to see this class information several times before it begins to make sense. Let’s get going on the first pass through this material.

**Figure 7-3**

Two ways to create Date objects.

class declaration

block of code that contains data and prototypes, defining the class

The **class declaration** is a block of source code that contains the keyword “class” followed by the name of the class. The name of the class is a programmer-defined data type. The block of code has data declarations and function prototypes enclosed within a set of braces {}. A semi-colon follows the closing brace. The class declaration can be thought of as the job description or a blueprint for the object.

```
class ClassName           //the ClassName is a programmer-defined data type
{
    //data declarations and function prototypes
    //go inside the class { }
};                         //don't forget the ;
```

class members

either data or functions that are declared within a class

The data and functions contained inside the class are known as **class members**. The class data represents the data that the object needs. Functions are the tasks that the object performs. A wonderful feature of the C++ classes is that the class data is seen by the class functions. This means that class functions can see/use/change the class data—no need to pass class data between class functions!

For now, glance over the entire class declaration for the Date (seen in Program 7-1, page 378). It contains several things we haven’t seen before, and frankly, it looks a bit intimidating. We’ll begin by isolating the code segments and discussing their purpose. Flip back and forth between the discussion and the actual Date declaration. Note: the **class definition** is the portion of the source code where the class functions are actually written.

We use the keyword “class” and the name Date, followed by an opening brace {. We are creating the Date class “job description” (or blueprint) within this block of code. This textbook will always capitalize our class names, i.e., “Date”, so that you’ll recognize it as a class. This is a common convention that C++ (and Java) programmers use. At the end of the declaration is a closing brace and semi-colon };

class definition

the source code that contains the implementation of member functions

```
class Date
{
    //Date class members are in here
    //Class members consist of data variables and function prototypes.
};
```

Before we move on, notice how this Date declaration is contained in a file named Date.h.

//File: Date.h Date Class Declaration

Many programmers follow the C++ convention where the name of the class dictates the name of the *.h header file that contains it. The class function bodies are in a *.cpp file of the same name, such as Date.cpp. (Note: functions that contain only one line of code can be written in the class declaration. See the below code.) This convention results in each class' source code being contained in a pair of files, such as Date.h and Date.cpp. It makes code re-use a snap, as you merely move the pair of class files into your project and go!

Back to the code—inside the class, you'll see a private and public section.

```
class Date
{
private:
    //data and function prototypes
public:
    //data and function prototypes
};
```

The keywords **private** and **public** are **access specifiers**, which dictate the accessibility of the members to the “world.” Any area of the program outside the class cannot directly access private class members. The public members are accessible via the object. This probably doesn't make any sense to you right now. That's OK, just be aware that some class members are available to the “world” (public) and some are not (private). Keep reading! More on access specifiers soon!

Let's look at the data declarations. Our Date class needs to have the data for one date. Therefore, inside the private section of the class we see these declaration statements for our class data:

```
class Date
{
private:
    int month, day, year;
    string description;
    int dayOfYear;
    bool bLeap;
```

Here we have defined three integers to hold the date numbers (i.e., 3, 14, 2007). We have a string for the description (i.e., “Jason's Birthday”), an integer for the day of the year (i.e., 73) and a boolean flag for the leap year (2007 is not a leap year, so it would be false). You might think that we are missing the name of the month, (i.e., “March”). In this design, we opt to have the name as a local class function variable

private

private class members are only seen by members within the class

public

public class members are accessible to all class members as well as by an object of the class

access specifiers

dictates the accessibility of class members to the program

instead of a class variable. We need the name only when we put the formatted string together, so we'll keep it inside the function. (We'll examine this soon.) When you design a class, you have to decide the necessary data the class needs. Often functions contain other local variables that aren't needed any place else.

If you go back and review Figures 7-1 and 7-2, they show that we can get information from our object or set information into our object. Remember, we can ask our object for the date values and a formatted string, as well as set new date data into the object. Look at these prototypes within the public section of the class. These prototypes define these get and set tasks:

```
public:
//Here are the Set functions to pass data into the object.
void SetDate(int m, int d, int y, string desc);
void SetDesc(string d){ description = d; }

//This Get function returns the formatted date string.
string GetFormattedDate();

//These Get functions return the integer values of class data.
int GetDayOfYear(){ return dayOfYear;}
int GetYear(){ return year;}
int GetMonth(){ return month;}
int GetDay(){ return day;}

//This function returns true/false concerning leap year status.
bool isLeapYear(){ return bLeap; }
```

Note: This is a new format for several of these functions. If a class function contains many lines of code, then the function body is located in the Date.cpp file. If a function contains one line of code, you can write it inside the class declaration like we have done here. The *GetYear* function:

```
int GetYear(){ return year;}
```

Compare this one-line function with the more conventional way to write a function. These two *GetYear* functions are equivalent.

<pre>int GetYear(){ return year; }</pre>	<pre>int GetYear() { return year; }</pre>
--	---

There are two function prototypes inside the private section of the class.

```
class Date
{
private:
    //data declaration are here
    //two function prototypes in the private class section
    void CalcDayOfYear();
    void DetermineLeapYear();
```

Remember we asked you, “Who’s job is it to calculate the day of the year and leap year status?” Well, the answer is that it is the Date class’ job to do this. These two functions have the necessary code for these tasks. The *DetermineLeapYear()* function sets the boolean *bLeap* value to true or false. The *CalcDayOfYear()* function counts the days and assigns the value into the *dayOfYear* integer. These two functions are located in the private class section because we are planning to call them from inside other class functions. It is not the object’s job to call these functions.

In the public section of the class declaration you’ll see two functions that have the same name as the class:

```
class Date
{
private:
    //private members are here
public:
    //The Date() functions are the class constructors.
    Date();
    Date(int m, int d, int y, string desc);
```

These functions are known as the **class constructor functions**. The class constructor is executed automatically when an object is created. Class constructors do not have a return type and may be overloaded. An **overloaded class constructor function** requires inputs be passed into the object on creation. We’ll see an example of an overloaded constructor shortly. Figure 7-4 shows how we designed our Date class so that a programmer could create a Date object with default date data (today’s date), or so that we can pass our own data into the object. The constructor with no inputs results

class constructor function

class member called automatically when an object is created

overloaded class constructor function

a constructor that requires input values

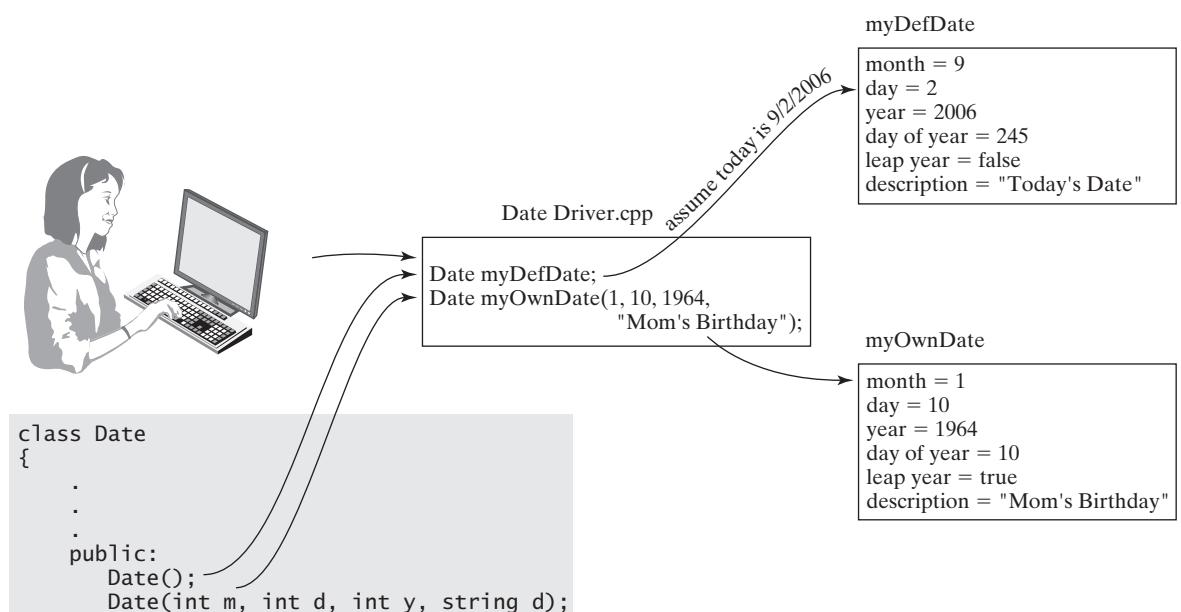


Figure 7-4

Our Date class provides the programmer with two ways to construct objects.

in a date with our default data (in the Figure, we assume 9/2/2006 is today's date). The constructor with four input values allows us to pass our data into the object.

The last new feature in our header file is seen near the top and bottom of the file. We see `#ifndef` and `#define` statements at the top, and a `#endif` statement at the bottom of the file:

```
#ifndef _DATE_H
#define _DATE_H

#include <string>
using namespace std;

class Date
{
//class members
};

#endif
```

pre-processor directive

a statement that provides instructions to the compiler

These new statements are ***pre-processor directive*** statements. A pre-processor directive statement provides instructions for the compiler. “Appendix H, Multi-file Programs” explains the purpose of these statements in detail. Basically, once we begin writing classes, we need to include our own class header *.h files in several other source code files. We don't want the compiler to read a header file more than once because it will think you are redefining the class. (Compiler errors!) The line `#ifndef _DATE_H` is telling the compiler, “If you do not know this block of code we're calling “`_DATE_H`”, then `#define _DATE_H` (i.e., define the block) between the `#define` and `#endif` to be “`_DATE_H`”. If the compiler sees the `#ifndef` statement again, it will know what “`_DATE_H`” is, and skip over it.

Here is the complete Date.h header file. We'll be using our Date class for several program examples in this chapter. Look over it once again.

Program 7-1

```
1 //File: Date.h  Date Class Declaration
2
3 #ifndef _DATE_H
4 #define _DATE_H
5
6 #include <string>
7 using namespace std;
8
9 class Date
10 {
11     private:
12         int month, day, year;
13         string description;
14         int dayOfYear;
15         bool bLeap;
16         void CalcDayOfYear();
```



```
17     void DetermineLeapYear();  
18  
19     public:  
20         Date();  
21         Date(int m, int d, int y, string desc);  
22         void SetDate(int m, int d, int y, string desc);  
23         void SetDesc(string d){ description = d; }  
24  
25         string GetFormattedDate();  
26  
27         int GetDayOfYear(){ return dayOfYear; }  
28         int GetYear(){ return year; }  
29         int GetMonth(){ return month; }  
30         int GetDay(){ return day; }  
31         bool isLeapYear(){ return bLeap; }  
32     };  
33  
34 #endif
```

Let's move on now to using our Date class. You may be thinking, "HEY! We haven't written the rest of the class yet!" You are right, we haven't, but it's more fun to eat dessert before dinner! Let's see how we can use our own class in a program. (Yes, of course, you must write the class in order to use it!) You are already a pro at using the C++ classes that are a standard part of the language. Using a class that we have written follows the same rules. Notice that above *main()* we include the Date.h file. This header file defines the Date class. We make two objects (named *myDefDate* and *myOwnDate*) and use them to call the *GetFormattedMessage()* function. Remember that our class is designed so that if we make a Date object with no inputs the default date will be today's date.

Here is the main function for the DateDemo Program. Let's look at the source code and output.

Program 7-1

```
1 //File: DateDriver.cpp  
2  
3 #include <iostream>  
4 #include "Date.h"  
5  
6 using namespace std;  
7  
8 int main()  
9 {  
10     cout<<" The first demo program for our Date class. \n";  
11  
12     //Make one Date object using the default constructor.  
13     //It contains our default data—which is today's date.  
14     Date myDefDate;  
15 }
```



```

16      //Make another Date object and pass in our own data.
17      Date myOwnDate(1,10,1964,"Mom's Birthday");
18
19
20      //Now ask each Date object for its formatted string.
21      string sDefDate = myDefDate.GetFormattedDate();
22      string sOwnDate = myOwnDate.GetFormattedDate();
23
24      cout << " The Default Date is: \n ";
25      cout << sDefDate << endl;
26
27      cout << " My Own Date is: \n ";
28      cout << sOwnDate << endl;
29
30      return 0;
31  }

```

Output

The first demo program for our Date class.
The Default Date is:
Today's Date: September 2, 2006
My Own Date is:
Mom's Birthday: January 10, 1964.

WOW! Isn't that great? It's almost like magic! In main, we made a Date object named “*myDefDate*.” By default, the date will be set to today's date. Notice that we didn't pass anything to the object when we created it. The Date object named “*myOwnDate*” is passed the data for Mom's Birthday.

```

Date myDefDate;
Date myOwnDate(1,10,1964,"Mom's Birthday");

```

The string we received from the *myDefDate* object contained “Today's Date: September 2, 2006” and the *myOwnDate* object passed back the “Mom's Birthday: January 10, 1964”.

```

//Now ask each Date object for its formatted string.
string sDefDate = myDefDate.GetFormattedDate();
string sOwnDate = myOwnDate.GetFormattedDate();
cout << "\n The Default Date is: \n ";
cout << sDefDate << endl;

cout << "\n My Own Date is: \n ";
cout << sOwnDate << endl;

```

Program 7-2 is the named OneDate, and we just create one default date object. This program illustrates how the various public Date class functions are called. Notice how we ask the user for a new string description then set it into the object.

Program 7-2

```
1 //File: OneDateDriver.cpp
2
3 #include <iostream>
4 #include "Date.h"
5
6 using namespace std;
7
8 int main()
9 {
10     cout<<" The second demo program for our Date class. \n";
11
12     //Use default constructor to make a date object.
13     Date myDate;
14
15     //Now ask Date object for its formatted string.
16     string sDate = myDate.GetFormattedDate();
17     cout << "The formatted date is: \n ";
18     cout << sDate;
19
20     //get individual values from Date object
21     cout << "\n     month = " << myDate.GetMonth();
22     cout << "\n     day = " << myDate.GetDay();
23     cout << "\n     year = " << myDate.GetYear();
24     cout << "\n day of year = " << myDate.GetDayOfYear();
25
26     //leap year?
27     cout << "\n " << myDate.GetYear();
28     if(myDate.isLeapYear() == true)
29         cout << " is a leap year" << endl;
30     else
31         cout << " is NOT a leap year" << endl;
32
33     //now ask user for a new description
34     string newDesc;
35     cout << "\n Enter a new description for today: ";
36     getline(cin,newDesc);
37
38     //set new description into the object
39     myDate.SetDesc(newDesc);
40
41     //write again
42     sDate = myDate.GetFormattedDate();
43     cout << "\n The date with new description is: \n ";
44     cout << sDate << endl;
45
46     return 0;
47 }
```



Output

The second demo program for our Date class.

The formatted date is:

Today's Date: September 2, 2006

month = 9

day = 2

year = 2006

day of year = 245

2006 is NOT a leap year

Enter a new description for today: First Lobo Football Game

The date with new description is:

First Lobo Football Game: September 2, 2006

In this program we use the other “get” functions to obtain the individual data items from the object. Notice how we make the call to the object within the *cout* statement. This technique is acceptable as long as the call to the class function is fairly simple.

```
//get individual values from Date object
    cout << "\n\n myDate Object info";
    cout << "\n     month = " << myDate.GetMonth();
    cout << "\n     day = " << myDate.GetDay();
    cout << "\n     year = " << myDate.GetYear();
    cout << "\n day of year = " << myDate.GetDayOfYear();
```

Next we asked the object if today’s date was a leap year. (Remember, it is the job of the class to determine if the year is a leap year.)

```
//leap year?
cout << "\n " << myDate.GetYear();
if(myDate.isLeapYear() == true)
    cout << " is a leap year" << endl;
else
    cout << " is NOT a leap year" << endl;
```

Lastly, we wish to change the description of this date, so we ask the user to enter a new description. We pass that to the object. Just to be sure it’s in there, we obtain the formatted string again, and print it to the screen.

```
//now ask user for a new description
string newDesc;
cout << "\n Enter a new description for today: ";
getline(cin,newDesc);

//set new description into the object
myDate.SetDesc(newDesc);

//write again
sDate = myDate.GetFormattedDate();
cout << "\n The date with new description is: \n ";
cout << sDate << endl;
```

The Secret Life of Classes

Do you own a car? How do you interact with your car? (No this isn't a trick question.) You use the key to start the car, step on the brake, and shift the car into gear (or put in the clutch to shift into gear). The steering wheel turns the front wheels—assisted by power steering. The gas pedal means go; the brake pedal means stop. The turn signal lever is on the left of the steering column and the windshield wipers lever is on the right. Headlights on and off by twist or pull of a knob. WOW. Easy.

The controls for your car are easy to use, easy to reach, and work in a logical manner. This result is not an accident. The car designers spend time thinking about how these controls are built and where they are placed. But "under the hood" your car is a very complicated piece of machinery! It is precisely constructed and computer controlled. The various parts are "hidden" from you. You do not know how the car stops when you step on the brakes, only that by stepping on the pedal, the car will stop. Automatic transmissions are very complicated, but all you have to do is put it into Drive and away you go!

Writing classes present a similar challenge to the C++ programmer. You want the public interface to the class to be easy to use, and you want to hide the inner workings of it. We don't know how the queue class keeps track of the items in a queue object, but we do know that if we use the *push()* function we can add an item to the end of it. The string class can find substrings within its data—but we don't know how it does it.

So far we've used classes by including a *.h file that contains their declarations, made an object and called its functions. We've seen our Date class declaration with its public and private members, but we've not seen how the Date object actually performs its tasks. It is time to examine the inner-workings of our Date class. The best way to start is to look at the declaration once again, to see the private and public members. Here is the Date declaration located in the Date.h file:

```
1  class Date
2  {
3  private:
4      int month, day, year;
5      string description;
6      int dayOfYear;
7      bool bLeap;
8      void CalcDayOfYear();
9      void DetermineLeapYear();
10
11     public:
12         Date();
13         Date(int m, int d, int y, string desc);
14         void SetDate(int m, int d, int y, string desc);
15         void SetDesc(string d){ description = d; }
16
17         string GetFormattedDate();
18
19         int GetDayOfYear(){ return dayOfYear; }
20         int GetYear(){ return year; }
```

```

21     int GetMonth(){ return month; }
22     int GetDay(){ return day; }
23     bool isLeapYear(){ return bLeap; }
24 };

```

The class members (the data and the functions/function prototypes within the braces) can see and access each other. This means that the class data is global to the class functions.

Recall that the last five functions in the Date declaration are “*Get*” functions, which return class members. They are one-line long and are defined right there. The *SetDesc()* is also defined there. It is passed the new description value, which it assigns to the class member. That leaves the other six functions to be written in the Date.cpp file.

Class functions that are written in the Date.cpp file must contain the class name with the scope operator (two colons). That is, if the class function is written (defined) outside of the Date declaration block of code (in the Date.h file) the **scope operator ::** is used to identify the class or the namespace who “owns” it. Writing the code this way says this function is a member of the Date class. See Figure 7-5.

We begin at the logical starting point, the class constructors. Recall that the class constructor function has the same name as the class. The class may have several constructor functions, i.e., they may be overloaded. The constructor does not have a return data type. It cannot return any value. The constructor function is executed automatically when an object is created. The main job of the constructor is to build an object and initialize that object’s data.

We have two Date constructor functions. One constructor is passed nothing, the other has four input values.

scope operator ::
used with a class name to indicate the class that “owns” the function, or the function is a member of that class

<p>Date.h (partial)</p> <pre> class Date { private: void CalcDayOfYear(); public: Date(); . . string GetFormattedDate(); . . }; </pre>	<p>Date.cpp (partial)</p> <pre> #include "Date.h" . . Date :: Date() { . . } void Date :: CalcDayofYear() { . . } string Date :: GetFormattedDate() { . . } </pre>
--	--

Figure 7-5

The class name and scope operator identify functions belonging to the Date class.

```

class Date
{
//private members
public:
    Date();           //default values constructor
    Date(int m, int d, int y, string desc); //overloaded

//rest of public members
};

```

We'll look at the easy one first. The overloaded constructor has four input values. Recall how we made a Date object and initialized the Date data to "Mom's Birthday" in Program 7-1:

```
Date myOwnDate(1,10,1964,"Mom's Birthday");
```

The overloaded constructor's code is located in the Date.cpp file. We need to use the class name and scope operator so that this constructor is identified as part of the Date class. The data values that are passed into the class are assigned to the appropriate class variables. We then call the private functions, *DetermineLeapYear()* and *CalcDayOfYear()* so that the *bLeap* and *dayOfYear* are determined.

```

Date::Date(int m, int d, int y, string desc)
{
    month = m;
    day = d;
    year = y;
    description = desc;
    DetermineLeapYear();
    CalcDayOfYear();
}

```



The Date constructor that sets the class data to the current date is a bit more complicated. We use functions from the ctime library to access the computer date. Before we get into the details of what we're doing, glance through this constructor to see how the month, day, year, and description are assigned, as well as the private calculate functions are called. Here is the code¹:

```

Date::Date()
{
    //Set the Date variables to the computer's date.
    time_t rawtime;
    tm *OStime;

    time(&rawtime);
    OStime = localtime(&rawtime);
    month = OStime->tm_mon + 1;
    day = OStime->tm_mday;
    year = OStime->tm_year + 1900;
}

```

¹In Microsoft Visual C++ 2005 localtime is flagged as deprecated and the programmer is advised to use localtime_s or to turn off the warning. A variety of internet websites maintain that localtime is still safe to use and should be used. The student may investigate the implementation of localtime_s.

```

description = "Today's Date";
DetermineLeapYear();
CalcDayOfYear();
}

```

The `ctime` library gives us the tools we need to obtain the date and time from the computer's clock. In a nutshell, we need a `time_t` variable and a pointer to a `tm data struct` variable. (A `data struct` is similar to a class with public data and without the class functions.) The `time()` function is called, filling a raw time value from the operating system. The `localtime()` function can then translate that raw time into a `data struct`, containing the values for the current month, day, and year. (See the `Date.cpp` below for the complete illustration.)

Another way to picture a class is to imagine an area that is surrounded by a brick wall. Inside the wall are the private class variables. All class functions (public and private) can see and use them. Public functions are that part of the "job description" that we can ask an object to perform. See Figure 7-6.

When we write a class, we control how the class data is accessed and what we can do with the class data. In our programs, we make an object, and the object "comes equipped" knowing how to perform the class functions. We can tell the object to do any of the public functions! However, if we try and ask the object to perform a private class function, or to access the private data, the compiler will stop us, as we are not allowed to access the private members.

Let's examine Figure 7-7, which is a diagram of a `Date` object that was created with the no input constructor. The object is used to call the `GetFormattedString()` function. This function can access the class data and build the string.

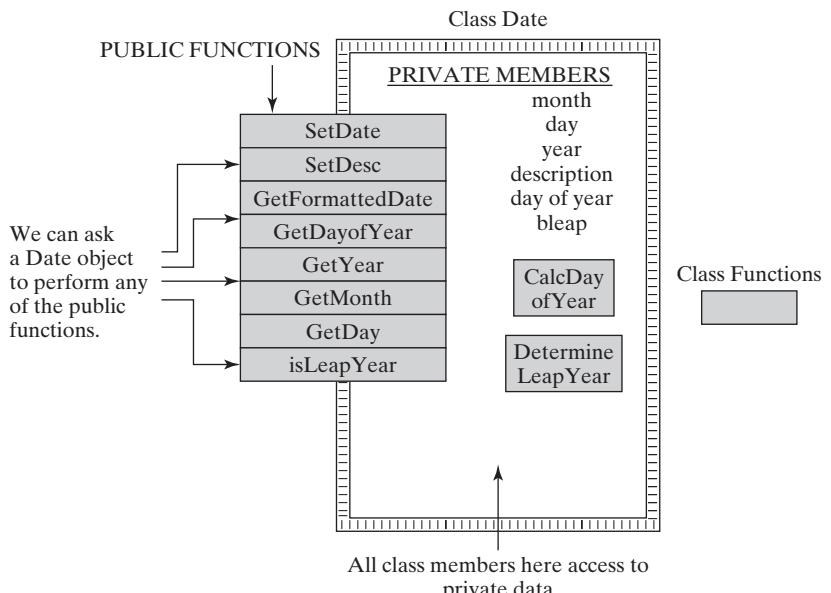
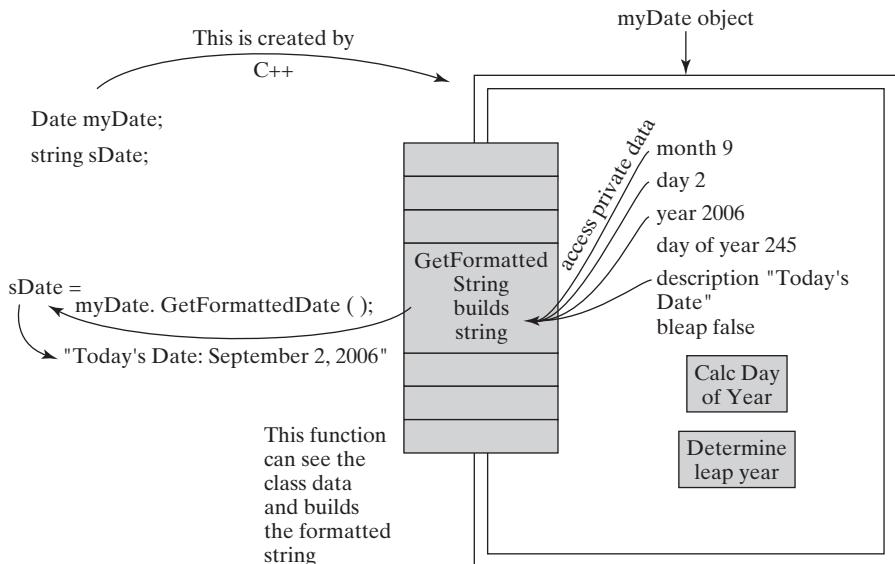


Figure 7.6

Visualize our `Date` class with private members inside a brick wall.

**Figure 7-7**

We can ask the `myDate` object to give us a formatted string describing its date data.

Here is the `Date` class' code for this function. It is contained inside the `Date.cpp` file. We first make a `stringstream` object and place the description into it. Next, we fill a string array with the month names and use the month value less one to obtain the month name. The day and year are added last.

```

string Date::GetFormattedDate()
{
    stringstream strDate;
    strDate << description;

    string monName[12] = {"January", "February ", "March",
                         "April", "May", "June", "July", "August",
                         "September", "October", "November", "December"};

    strDate << ":" << monName[month-1] << " " << day
        << ", " << year;
    return strDate.str();
}

```

Remember the question, “Who’s job is it to calculate the day of the year and determine the leap year?” The answer is the `Date` class. Now we look at how it accomplishes these two tasks. First, let’s make a `Date` object and then set new data into it.

```

Date myDate;           //use default constructor
//now change data to Ryan's birthday
myDate.SetDate(7,30,2004,"Ryan's Birthday");

```

Inside the `SetDate` function, the input data is assigned to the class members. Within the `SetDate()` function, the `DetermineLeapYear()` and `CalcDayOfYear()`

functions are called. All class functions can see and use/call other class functions directly. Here is the code:

```
void Date::SetDate(int m, int d, int y, string desc)
{
    month = m;
    day = d;
    year = y;
    description = desc;
    DetermineLeapYear();           //calls to private class functions
    CalcDayOfYear();
}
```

The *DetermineLeapYear()* function contains the algorithm for determining a leap year. This function's goal is to set the *bLeap* variable to either true or false. It is a *private* function and not passed any data, nor does it return any data. It merely works with the class' year value and sets *bLeap*.

```
void Date::DetermineLeapYear()
{
    //A year is a leap year if it is divisible by four,
    //unless it is a century date (i.e., 1900).
    //If it is a century date, it is a leap year only
    //if it is divisible by 400 (i.e., 2000).

    if(year%4 == 0 && year % 100 != 0)
        bLeap = true;
    else if(year % 400 == 0)
        bLeap = true;
    else
        bLeap = false;
}
```

The inner workings of the class control the calls to the private functions. The *CalcDayOfYear()* function relies on the *bLeap* variable being set correctly, so we need to be sure the *DetermineLeapYear()* is called before the *CalcDayOfYear()*. Inside this calculation function, we set up an array to hold the days of the months and run a *for* loop to count the days up to the previous month. If you were counting days for March 14, you'd count January + February days (if it is a leap year, add 1 more), then add the day for March.

```
void Date::CalcDayOfYear()
{
    //set up array of days in each month
    //for non-leapyear year
    int dayCount[12] = {31,28,31,30,31,30,
                        31,31,30,31,30,31};

    dayOfYear = 0;

    //add the days up to the previous month
```

```
for(int i = 1; i < month; ++i)
{
    dayOfYear += dayCount[i-1];
    //if adding Feb, check if leap year
    if(i == 2 && bLeap == true)
        dayOfYear += 1;
}
dayOfYear += day;
}
```

Below is the complete Date.cpp file. We used this code for the first two programs. Because we have now built a complete Date class, we'll use this class repeatedly in this chapter. Any future class or program we write that needs a Date, we are ready to go! Just drop the Date.cpp and Date.h files into the project folder, include Date.h where needed and we're done! We have a Date.

Program 7-1 and 7-2 Date.cpp File

```
1 //File: Date.cpp
2
3 #include <iostream>
4 #include <ctime>           //obtain system date
5 #include <iomanip>
6 #include "Date.h"
7 using namespace std;
8
9 Date::Date()
10 {
11     //Set the Date variables to the computer's date.
12     time_t rawtime;
13     tm *OStime;
14
15     //First obtain the raw time from the O/S system
16     time(&rawtime);
17
18     //localtime converts this to data struct
19     //containing month, day, year
20     //Jan month is 0, must add 1
21     //0 year is 1900,
22     //must add 1900 to obtain correct year
23     OStime = localtime(&rawtime);
24
25     month = OStime->tm_mon + 1;
26     day = OStime->tm_mday;
27     year = OStime->tm_year + 1900;
28
29     description = "Today's Date";
30     DetermineLeapYear();
31     CalcDayOfYear();
32 }
```



```
33
34 Date::Date(int m, int d, int y, string desc)
35 {
36     month = m;
37     day = d;
38     year = y;
39     description = desc;
40     DetermineLeapYear();
41     CalcDayOfYear();
42 }
43
44 void Date::SetDate(int m, int d, int y, string desc)
45 {
46     month = m;
47     day = d;
48     year = y;
49     description = desc;
50     DetermineLeapYear();
51     CalcDayOfYear();
52 }
53
54 string Date::GetFormattedDate()
55 {
56     stringstream strDate;
57     strDate << description;
58
59     string monName[12] = {"January", "February ", "March",
60                         "April", "May", "June", "July", "August",
61                         "September", "October", "November", "December"};
62
63     strDate << ":" << monName[month-1] << " " << day
64             << ", " << year;
65
66     return strDate.str();
67 }
68
69 void Date::CalcDayOfYear()
70 {
71     //set up array of days in each month
72     //for non-leapyear year
73     int dayCount[12] = {31,28,31,30,31,30,
74                         31,31,30,31,30,31};
75
76     dayOfYear = 0;
77
78     //add the days up to the previous month
79     for(int i = 1; i < month; ++i)
```

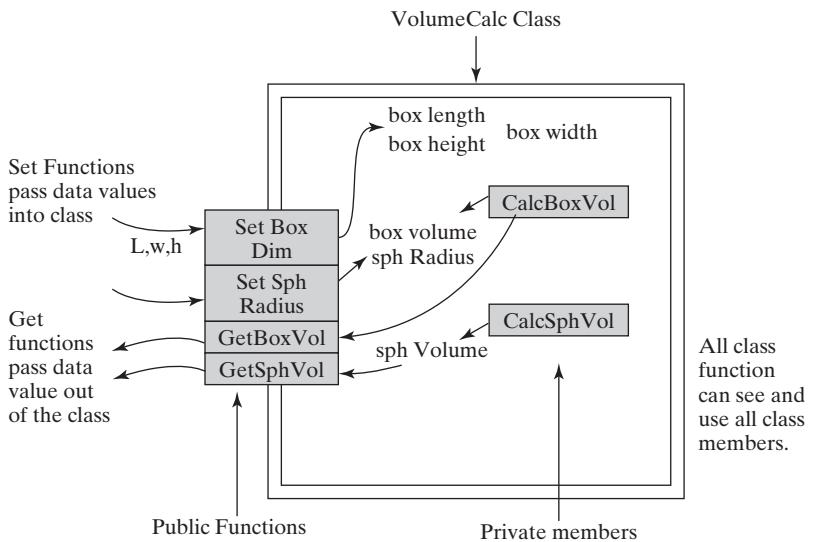
```
80     {
81         dayOfYear += dayCount[i-1];
82
83         //if adding Feb, check if leap year
84         if(i == 2 && bLeap == true)
85             dayOfYear += 1;
86     }
87
88     dayOfYear += day;
89 }
90
91 void Date::DetermineLeapYear()
92 {
93     //A year is a leap year if it is divisible by four,
94     //unless it is a century date (i.e., 1900).
95     //If it is a century date, it is a leap year only
96     //if it is divisible by 400 (i.e., 2000).
97
98     if(year%4 == 0 && year % 100 != 0)
99         bLeap = true;
100    else if(year % 400 == 0)
101        bLeap = true;
102    else
103        bLeap = false;
104 }
```

The Importance of *set* and *get* Functions and the Volume Calculator Class

In the world of object-oriented programs, the use of “*get*” and “*set*” functions is very important. Class data is normally placed in the private section of the class, which means that the “world” cannot access them directly. In order to pass data into the class, or get data out of the class, we use the *set* and *get* functions.

Our next program is a simple one. There is a volume calculator class named *VolumeCalc*, and its job is to calculate the volume of a box and/or a sphere. Imagine that we have Bob, our *VolumeCalc* object. He is a pro at calculating box volume or sphere volume, but before he can do his job, what does he need to know? I hope you are thinking the box dimensions or the sphere’s radius. For Bob to do his job, we need to pass the box and sphere values to him using his *set* functions and let him do the calculations. Then, when we need the volume information, we call Bob’s *get* functions to get the data from the class. Figure 7-8 illustrates how the public *set* and *get* functions are the gateways for data flowing into and out of the class.

If you peruse object-oriented documentation for C++ (or Java), most classes have *set* and *get* functions. The sets always pass data into the class. The gets always obtain data from the class. For now, examine Program 7-3 here.

**Figure 7-8**

The *set* functions are used to pass data into the class and *get* functions return data from a class.

Program 7-3

```

1  #ifndef _VOLCALC_H
2  #define _VOLCALC_H
3
4  #include <string>
5  using namespace std;
6
7  class VolumeCalc
8  {
9  private:
10     double sphRadius;
11     double boxLength, boxWidth, boxHeight;
12     double sphVolume, boxVolume;
13
14     void CalcBoxVol(); //called from the set
15     void CalcSphVol(); //functions
16
17 public:
18     VolumeCalc(); //constructor
19
20     void SetSphRadius(double r);
21     void SetBoxDim(double l, double w, double h);
22
23     double GetSphVol(){ return sphVolume; }
24     double GetBoxVol(){ return boxVolume; }
25
26 };
27
28 #endif

```

In the *main* function we create our *VolumeCalc* object named Bob. We then ask the user to enter the values for the box and sphere. We call the appropriate *set* functions. Bob automatically calculates the volumes for us. We then call the *get* functions for our answers.

Program 7-3

```
1 //File: CalcDriver.cpp
2 //This program uses a VolumeCalc object
3 //to compute them volume of a box and sphere.
4
5 #include "VolumeCalc.h"
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     cout <<"\n The Volume Calculator Program"
12         << "\n Calculates the volume of a "
13         << " box or a sphere." << endl;
14
15     VolumeCalc Bob;           //make a VolumeCalc object
16
17     //we'll have Bob do the calculations for us. :-) Go Bob!
18
19     //local vars for user to input data
20     double len, wid, hgt, rad;
21
22     cout << "\n Box: Enter the length width "
23         " and height \n\n ";
24     cin >> len >> wid >> hgt;
25
26     cout << "\n Sphere: Enter the radius ";
27     cin >> rad;
28
29     //Now give Bob the box and sphere dimensions.
30     //We set the data into the object.
31     Bob.SetBoxDim(len,wid,hgt);
32
33     Bob.SetSphRadius(rad);
34
35     //Now we ask Bob to give us the volumes.
36     //We use the get functions.
37     double sphereV = Bob.GetSphVol();
38     double boxV = Bob.GetBoxVol();
39
40     cout.setf(ios::fixed);
41     cout.precision(3);
42
```



```

43     cout << "\n Box Volume: " << boxV
44     << "\n Sphere Volume: " << sphereV;
45
46     cout << "\n\n Thanks to Bob, our VolumeCalc object!\n";
47
48     return 0;
49 }
```

Output

The Volume Calculator Program
Calculates the volume of a box or a sphere.

Box: Enter the length width and height
10.6 8.5 5.7

Sphere: Enter the radius 11.2
Box Volume: 513.570
Sphere Volume: 525.442
Thanks to Bob, our VolumeCalc object!.

The class functions are short. Notice how the two *set* functions call the private *CalcVol* functions right after the data values are assigned into class members. It is not the user's job to tell Bob to do the calculations. It is Bob's job to calculate the values once he has the necessary data. The *Get* functions (seen in VolumeCalc.h file above) are one-liners, because their job is to just return the volume values.

Program 7-3

```

1 //File: VolumeCalc.cpp
2 //Contains some of the function definitions
3 //for the VolumeCalc class.
4
5 #include "VolumeCalc.h"
6 #include <cmath>
7 using namespace std;
8
9 VolumeCalc::VolumeCalc()          //constructor
10 {
11     boxHeight = boxWidth = boxLength = 0.0;
12     sphRadius = 0.0;
13     boxVolume = sphVolume = 0.0;
14 }
15
16 //Once the set functions have been passed data,
17 //they call the calculate functions.
18
19 void VolumeCalc::SetSphRadius(double r)
20 {
21     sphRadius = r;
```

```
22     CalcSphVol();  
23 }  
24  
25 void VolumeCalc::SetBoxDim(double l, double w, double h)  
26 {  
27     boxLength = l;  
28     boxWidth = w;  
29     boxHeight = h;  
30     CalcBoxVol();  
31 }  
32  
33 //The calculate functions do their math and  
34 //assign the results into the class variables.  
35 //The get functions return them.  
36 void VolumeCalc::CalcBoxVol()  
37 {  
38     boxVolume = boxLength * boxWidth * boxHeight;  
39 }  
40  
41 void VolumeCalc::CalcSphVol()  
42 {  
43     double PI = 3.14159265;  
44     sphVolume = 4.0/3.0*PI*pow(sphRadius,2);  
45 }
```

PICalculator Class

Lets's write another class and use the object in a program! This time we'll rewrite the PI Calculator as a class. We saw it in Chapter 3 and again in Chapter 4. Glance back at Program 4-13 (page 221) to see how we used standalone functions to perform PI related jobs for us. Now we'll write a class to handle all these programmatic details for us.

The PICalculator class is contained in the PICalculator.h file. The private members include an integer for the number of terms and two doubles for the calculated and actual value of PI. We'll call our *Calculate* function from public members. The one constructor function sets the actual value of PI and zeros out the number of terms variable. Remember, flexibility is the key to writing good classes. We'll provide two different ways to tell the *PICalculator* object how many terms we wish to use. The *SetNumTerms()* is passed the integer value for terms. The *AskNumTerms()* function asks the user to enter the number of terms. Both of these functions call the private *Calculate()* function. The three *Get* functions allow our user to obtain numeric and string data. Examine the files for Program 7-4. The output is seen after the PIDriver.cpp file.

Program 7-4

```
1 //File: PICalculator.h  
2
```



```

3  //Given a number of terms, this PICalculator
4  //class uses an infinite series to determine
5  //an estimate for PI.
6
7  #include <string>
8  using namespace std;
9
10 #ifndef _PICALC_H
11 #define _PICALC_H
12
13 class PICalculator
14 {
15     private:
16         int nTerms;
17         double calcPI, realPI;
18
19         //This performs the PI calculation.
20         void Calculate();
21     public:
22         PICalculator();
23
24         //Ask user for number of terms.
25         void AskNumTerms();
26
27         //Set function passes data into object
28         void SetNumTerms(int tms);
29
30         //Get functions return values.
31         double GetActualPI(){ return realPI;}
32         double GetCalcdPI(){ return calcPI;}
33         string GetResultsString();
34     };
35
36 #endif

```

The PICalculator.cpp file contains the multi-line class functions. Notice how the *Calculate()* function is called whenever the object obtains the number of terms.

Program 7-4

```

1  //File: PICalculator.cpp
2
3  #include "PICalculator.h"
4  #include <string>
5  #include <iostream>
6  #include <iomanip>           //for setprecision
7  #include <sstream>           //for a stringstream object
8  using namespace std;
9

```

```
10 //constructor initializes class members
11 PICalculator::PICalculator()
12 {
13     nTerms = 0;
14
15     //Initialize real value of PI to 16 places
16     realPI = 3.1415926535897932;
17 }
18
19 //use number of terms to determine est PI
20 void PICalculator::Calculate()
21 {
22     calcPI = 0.0;
23     double numerator = 4.0, denom = 1.0;
24
25     for(int i = 0; i < nTerms; ++i)
26     {
27         calcPI = calcPI + numerator/denom;
28         denom += 2.0;
29         numerator = -1.0 * numerator; //flip sign
30     }
31 }
32
33 //Ask the user for number of terms.
34 void PICalculator::AskNumTerms()
35 {
36     cout << "\n Enter number of terms for PI calculation: ";
37     cin >> nTerms;
38     cin.ignore(); //strip off Enter key
39
40     //now calculate PI
41     Calculate();
42 }
43
44 //Set the number of terms and calculate.
45 void PICalculator::SetNumTerms(int tms)
46 {
47     nTerms = tms;
48
49     //Once you set the number of terms,
50     //calculate the value for PI.
51     Calculate();
52 }
53
54 //Return a formatted string with results.
55 string PICalculator::GetResultsString()
56 {
57     stringstream piString;
```

```
59     piString.setf(ios::fixed);
60     piString << "\n Results\n Number of Terms: " << nTerms
61         << setprecision(10) << " PI = " << realPI
62         << " Calc PI = " << calcPI << endl;
63
64     return piString.str();
65 }
```

In our *main* function we test both techniques for telling the object how many terms to use. We make Steve, our *PICalculator* object. We tell Steve to use 5000 terms and obtain the results. We then allow Steve to ask the user how many terms he wishes to use. It is a nice feature to provide both ways, but don't think that you must use both in a program. (Sometimes students feel that they must use every function they write!) Here it is easy to code, easy to use, and ready to go either way.

Program 7-4

```
1 //A program that uses the PICalculator object
2 //to calculate an estimated value of PI.
3
4 #include "PICalculator.h"
5 #include <iostream>
6 #include <string>
7 using namespace std;
8
9 int main()
10 {
11     //Steve is our PICalculator object.
12     //He'll help us calculate values of PI.
13     PICalculator Steve;
14
15     string result;
16
17     cout << "\n Revisit the PI Calculation Program\n";
18
19     //First we'll tell Bob that we want to
20     //calculate PI to 5000 terms.
21
22     int terms = 5000;
23     Steve.SetNumTerms(terms);
24
25     //Ask Steve to give us the results!
26     result = Steve.GetResultsString();
27
28     cout << "\n We told Steve to use 5000 terms: " << endl;
29     cout << result;
30
31     //Now we let Steve ask the user for the
32     //number of terms.
```

```
33     Steve.AskNumTerms();
34     result = Steve.GetResultsString();
35
36     cout << "\n The user told Steve how many terms: "<< endl;
37     cout << result << endl;
38
39     return 0;
40 }
```

Output

Revisit the PI Calculation Program

We told Steve to use 5000 terms:

Results

Number of Terms: 5000 PI = 3.1415926536 Calc PI = 3.1413926536

Enter number of terms for PI calculation: 5000000

The user told Steve how many terms:

Results

Number of Terms: 5000000 PI = 3.1415926536 Calc PI = 3.1415924536

7.3

Objects as Class Members

Now that we've begun to write our own classes, it is time to see how we write a class that contains an object as part of its class data. It is reasonable to expect that our classes will make use of other classes. By writing a class that contains an object, it really helps us learn how to use our objects. This idea of having a class contain another object isn't new. If you look at our Date class, it contains a string object for the description. To this point we have used local objects within our class functions. Look at the Date's *GetFormattedDate* function. We made a *stringstream* object and used it to build our formatted string. Now we're going to see how to have an object is a member of a class.

The Have You Had Your Birthday? Program

In Program 7-5 we'll ask the user to enter his name and birth date. The program calculates his age and whether he has had his birthday. The program determines either how many days it has been since his birthday, or how many days it will be until his birthday. If his birthday is today, it wishes him Happy Birthday! The Person class incorporates two Date objects as part of its class data. We'll use of several of the Date functions.

For this program we'll create the Person class that contains private data for the Person's name, age, and Date objects for the birthday and today's date. We'll use today's date for calculating the age and counting days to/from the birthday. We'll keep it simple to use by having our *Person* object call only two functions—*AskForBDayInfo()* and *WriteBDayInfo()*. The other class functions, *CalcDaysToFromBDay()* and *CalculateAge()* will be called from within the *Ask* function. Here is the class declaration for our Person.



```
Program 7-5
1 //File: Person.h
2
3 #ifndef _PERSON_H
4 #define _PERSON_H
5
6 #include <string>
7 #include "Date.h"
8 using namespace std;
9
10 class Person
11 {
12     private:
13         string name;
14         Date bday, today;
15         int age;      //person's age in years
16
17         //number of days to/from birthday
18         int daysToFromBday;
19
20         //birthday yet this year?
21         //based on today's date,
22         bool bHadABirthday;
23
24         void CalcDaysToFromBday();
25         void CalculateAge();
26
27     public:
28         Person();
29         void AskForBDayInfo(); //Calls CalcAge() & CalcDays()
30         void WriteBDayInfo();
31     };
32
33 #endif
```

The *main* function is very straightforward, to the point of being boring. We'll make a *Person* object, call the *Ask* and *Write* functions within a do *while* loop. Who's job is it to determine the age and birthday status? Obviously, it is the *Person* class. Remember, once the name and age have been entered into the object, it is time to determine these facts. We do not want to make our programmer who is using this class have to make the calls to the calculation functions. That is why the functions are private. Once the object gets the data—have your class functions call them automatically!

```
Program 7-5
1 //File: Driver.cpp
2
3 #include "Person.h"
```

```
4 #include <iostream>
5 using namespace std;
6
7 int main()
8 {
9     string goAgain;
10    Person user;
11
12    cout<<"\n This is the "
13    << "\\"Have You Had Your Birthday?"\\ Program!";
14
15    //For convenience, let's write out today's date.
16    Date d;
17    cout << "\n " << d.GetFormattedDate();
18
19    do
20    {
21        user.AskForBDayInfo();
22        user.WriteBDayInfo();
23
24        cout<<"\n\n Would you like to go again? yes/no ";
25        getline(cin, goAgain);
26
27    }while(goAgain == "yes");
28
29    cout<<"GoodBye";
30
31    return 0;
32 }
```

Output

This is the "Have You Had Your Birthday?" Program!

Today's Date: September 3, 2006

What is your name? Hannah

Hannah, when is your birthday (M/D/Y)? 8/25/1995

Hannah's Birthday: August 25, 1995 makes him/her 11 years old.

Hannah had his/her birthday 9 days ago.

Would you like to go again? yes/no yes

What is your name? Meg

Meg, when is your birthday (M/D/Y)? 9/3/1999

Meg's Birthday: September 3, 1999 makes him/her 7 years old.

Happy birthday!!!

Would you like to go again? yes/no no.

The exciting part of this program can be found on the inside of the Person.cpp file! The *Ask()* function obtains the name and date data, setting info

into the *bdy Date* object. We then call the two private *Person* functions that determine birthday status and age. Our *Date* objects can tell us the day of year for the data, so it is easy to take the difference and check the sign to determine if and when the birthday occurred. The *Write()* function uses the Date formatted string to aid in writing results. Read through the code and see how the Person class utilizes the *Date* object tools.

Program 7-5

```
1 //File: Person.cpp
2
3 #include "Person.h"
4 #include <string>
5 #include <iostream>
6 using namespace std;
7
8 Person::Person()
9 {
10     name = "";
11     age = 0;
12     daysToFromBday = 0;
13     bHadABirthday = false;
14 }
15
16 void Person::CalcDaysToFromBday()
17 {
18     //We need the day of year for the
19     //bdy and today. Problem is if the
20     //person was born in a leapyear,
21     //our diff would be one day off.
22     //For example:
23     //Mar 1, 2000 = 61st day
24     //Mar 1, 1999 = 60th day
25
26     //To solve, we'll make our own date using
27     //today's month & day, and bdy's year.
28     //We use the Date's overloaded constructor.
29     //We pass in a "" string, since it won't be used.
30     Date check(today.GetMonth(), today.GetDay(),
31                 bday.GetYear(), "");
32
33     //Now take the difference from bdy and today
34     int diff;
35     diff = bday.GetDayOfYear() - check.GetDayOfYear();
36
37     //if the diff is 0 or negative
38     //then the person already had a Bday
39     if(diff <= 0)
```



```
40     {
41         daysToFromBday = -1*diff;
42         bHadABirthday = true;
43     }
44 else
45 {
46     daysToFromBday = diff;
47     bHadABirthday = false;
48 }
49 }
50
51 void Person::CalculateAge()
52 {
53     age = today.GetYear() - bday.GetYear();
54
55     //This calc depends on knowing if the person has
56     //had a birthday. Need to be sure we have
57     //called CalcDays. Since these are both
58     //private functions, we are OK since we
59     //know from whence we came.
60
61     //if the person did not yet had a Bday
62     if(bHadABirthday == false ) age--;
63 }
64
65 void Person::AskForBDayInfo()
66 {
67     //ask for the name
68     cout<<"\n\n What is your name? ";
69     getline(cin,name);
70
71     //ask for the birthday
72     int d,m,y;
73     char slash;
74     cout <<"\n " << name
75     << ", when is your birthday (M/D/Y)? ";
76     cin >> m >> slash >> d >> slash >> y;
77
78     //remove newline from input queue
79     cin.ignore();
80
81     //set data into birthday object
82     string des = name + "'s Birthday";
83     bday.SetDate(m,d,y,des);
84
85     //Must call CalcDays function before the
86     //CalcAge function! See notes in CalcAge.
87     CalcDaysToFromBday();
```

```
88     CalculateAge();
89 }
90
91 void Person::WriteBDayInfo()
92 {
93     cout << "\n " << bday.GetFormattedDate() <<
94         " makes him/her " << age << " years old.";
95
96     //first check if it is his/her birthday
97     if(daysToFromBday == 0)
98     {
99         cout << "\n Happy birthday!!!" << endl;
100    return;
101 }
102
103 //not a birthday, so write results
104 if(bHadABirthday == true)
105 {
106     cout << "\n " << name << " had his/her birthday "
107         << daysToFromBday << " days ago.\n\n";
108 }
109 else
110 {
111     cout << "\n " << name
112         << " will have his/her birthday in "
113         << daysToFromBday << " days.\n\n" ;
114 }
115 }
```

7.4 Class Destructors

An object comes into existence when it is created and the constructor initializes the class variables. Just as the object is “born,” it also “dies” when the object goes out of scope or the program terminates. At program termination or when the object goes out of scope, the object is destroyed. Some programming situations call for the object to perform certain tasks when it is destroyed. These tasks may include closing a data file if the object had opened one, or freeing memory that the object had allocated. C++ provides a **class destructor function** that is called automatically (if it exists) when the object is destroyed. Global object destructors are called when the program terminates. If a function creates a local copy of an object, this local copy is destroyed when the function is exited.

class destructor function

class member called automatically when an object goes out of scope

The form of the destructor function is similar to that of a constructor function. The class name is used with a ~ in front of the name, and there is no return value. A class can have only one destructor function (unlike

constructors which may be overloaded.) The prototype of the destructor is shown below:

```
class ClassName
{
    public:
        ClassName();           //default constructor prototype
        ~ClassName();          //destructor function prototype, use the ~
};
```

We will see how the destructor is used in the Practice section of this chapter. For now, here is a simple program that instantiates three *Greetings* objects. This class contains a string for the name of who is being greeted, three constructors and one (and only one) destructor function. When the objects come into scope, the appropriate constructor is called. When the objects go out of scope (at program termination) the destructor is called for each. Look over the three files that make up the code in Program 7-6.

Program 7-6

```
1 //File: Greetings.h
2
3 #ifndef GREETING_H
4 #define GREETING_H
5
6 #include <string>
7 using namespace std;
8
9 class Greetings
10 {
11     private:
12         string name;
13
14     public:
15
16         //default constructor, writes greeting to Bob
17         Greetings();
18
19         //overloaded constructors
20         //writes standard greeting to name
21         Greetings(string n);
22
23         //writes greeting to name
24         Greetings(string g, string n);
25
26         //destructor
27         ~Greetings();
28 };
29
30 #endif
```

Notice that our *main* function is very short. The object constructor and destructor functions are called automatically. It is an interesting note that the first object created is the last one destructed. This is due to the order that the objects are stored on the stack—the first one created is the last one off.

Program 7-6

```
1 //DemoDriver.cpp
2 //A short program that demonstrates class
3 //destructor functions.
4
5 #include "Greetings.h"
6 #include <string>
7 using namespace std;
8
9 int main()
10 {
11     Greetings g1;
12
13     Greetings g2("Susie Q Frenchfry");
14
15     Greetings g3("Howdy", "Rodeo Bill");
16
17     return 0;
18 }
```

Output

```
Hello Bob from the default constructor!
Hello Susie Q Frenchfry! How are you today?
```

```
Howdy! Rodeo Bill
```

```
Goodbye Rodeo Bill from the Greetings destructor!
Goodbye Susie Q Frenchfry from the Greetings destructor!
Goodbye Bob from the Greetings destructor!
```

The Greetings.cpp file contains the function definitions for our class.

Program 7-6

```
1 //File: Greetings.cpp
2
3 #include "Greetings.h"
4 #include <string>
5 #include <iostream>
6 using namespace std;
7
8 Greetings::Greetings()
9 {
10     name = "Bob";           //By default, we set the name to Bob
11     cout << "\n Hello " << name << " from the default constructor! \n";
```

```

12 }
13
14 Greetings::Greetings(string n)
15 {
16     name = n;
17     cout << "\n Hello " << name
18     << "! How are you today? " << endl;
19 }
20
21 Greetings::Greetings(string g, string n)
22 {
23     name = n;
24     cout << "\n " << g << "! " << name << endl;
25 }
26
27 Greetings::~Greetings()
28 {
29     cout << "\n Goodbye " << name << " from the"
30     " Greetings destructor! \n";
31 }
```

7.5

Array of Objects

A C++ programmer may declare an array of objects in the same manner as he would declare an array of integers or an array of strings.

```

int nums[100];           //array of 100 integers
string names[25];        //array of 25 string objects
PhoneList myList[10];    //array of 10 PhoneList objects
```

The class needs to have a default constructor (i.e., one with no inputs) in order to create an array of objects. As with any array, the elements are accessed using an integer index. In our *nums* array, we use the *for* loop and *[i]* as part of the variable name. Here, we assign random numbers into the *nums* array:

```

for(int i = 0; i < 100; ++i)
{
    nums[i] = rand();           //assign random number into array elements
}
```

The same holds true for an array of objects. Via the object, we don't access the private data in the class, but we do call the public functions. If our *PhoneList* class has a public method for showing the listing, we call each object's function like this:

```

for(int i = 0; i < 10; ++i)
{
    myList[i].ShowListing();      //have each object write its info
}
```

Instead of seeing bits and pieces of program code, let's look at one complete program to see how easy it is to work with an array of objects. There is another example of an array of objects, *The Teacher's Helper*, in the Practice section of this chapter.

Array of PhoneList Objects

The *PhoneList* class represents a single entry for a telephone book listing, including strings for the name, address, and phone number. The data is private, so we must find a way to fill all the object variables. What functions do we need for our phone data? Hopefully, you were thinking *Set* functions! We'll use one-line sets and a function that prints the listing information to the screen.

Our class must have a default constructor in order to make an array of these objects. When C++ makes an array of objects, the program executes the default constructor for each object in the array. We'll create an array of five *PhoneList* objects. The constructor writes a message, which we see five times in our program output. The *main* function reads the phone numbers from a data file called *phone-book.txt*. (The file is read until it reaches the end of the file.) Once we've filled the array, we ask the objects in the array to write their data to the screen. See the three source code files that represent our Program 7-7. This program is illustrated in Figure 7-9.

Program 7-7

```
1 //File: PhoneList.h
2 //This class represents one phone entry.
3
4 #ifndef _PHONELIST_H
5 #define _PHONELIST_H
6
7 #include <string>
8 using namespace std;
9
10 class PhoneList
11 {
12     private:
13         string name, address;
14         string phNumber;
15     public:
16         PhoneList();
17         void SetName(string n) { name = n; }
18         void SetAddress(string addr) { address = addr; }
19         void SetNumber(string phN) { phNumber = phN; }
20         void ShowListing();
21
22     };
23
24 #endif
```

Class declaration

Each PhoneList object looks like this:

```
class PhoneList
{
    private:
        String name, address;
        String phNumber;
    public:
        PhoneList();
        void SetName(String n)
        { name = n; }
        void SetAddress(String addr)
        { address = addr; }
        void SetNumber(String phN)
        { phNumber = phN; }
        void ShowListing();
};
```

phonebook.dat

```
Dr. Frank N. Stein
Castle, The Old Country
555 111 9999
Count Dracula
1 Red Ave. Bloodsworth
555 111 8888
Little Miss Muffet
29 Tuffet St, Arachnoid
555 111 7777
Mr. Creature
Black Lagoon North
555 111 6666
Fred WereWolf
London, England
555 111 5555
```

Driver program

```
int main()
{
PhoneList myList[5];

/*after the data file is read,
each object is unique and
has different data*/
```

Five separate objects
are created with this
declaration.

mylist[0]

```
Dr. Frank N. Stein
Castle, The Old Country
555 111 9999
```

mylist[1]

```
Count Dracula
1 Red Ave. Bloodsworth
555 111 8888
```

mylist[4]

```
Fred WereWolf
London, England
555 111 5555
```

mylist[3]

```
Mr. Creature
Black Lagoon North
555 111 6666
```

mylist[2]

```
Little Miss Muffet
29 Tuffet St, Arachnoid
555 111 7777
```

Figure 7-9
PhoneList array
and the five
objects.

Program 7-7

```
1 //File: ArrayOfObjects.cpp
2 //Array of Objects Program
3 //This program uses an array of PhoneList objects
4 //It read 5 entries from a data file.
5
6 #include <iostream>
7 #include <string>
8 #include <fstream>
9 #include "PhoneList.h"
10 using namespace std;
11
```

```
12 #define FILE_IN "phonebook.txt"
13
14 int main()
15 {
16     //make an array of 5 PhoneList objects
17     PhoneList mylist[5];
18
19     //input stream for reading from file
20     ifstream input;
21
22     //open input
23     input.open(FILE_IN, ios::in);
24
25     //Check to be sure file is opened.
26     if(!input)
27     {
28         cout << "\n Can't find input file " << FILE_IN;
29         cout << "\n Exiting program, bye bye \n ";
30         exit(1);
31     }
32
33     string buffer;
34     int count = 0;
35
36     //read until we reach end of file
37     //use set functions for each object
38     while( !input.eof() )
39     {
40         //read name
41         getline(input,buffer);
42         mylist[count].SetName(buffer);
43
44         //read address
45         getline(input,buffer);
46         mylist[count].SetAddress(buffer);
47
48         //read number
49         getline(input,buffer);
50         mylist[count].SetNumber(buffer);
51
52         ++count;
53     }
54
55     cout.setf(ios::left); //left justify output
56
57     cout << "\n\n Your Address Book contains these "
58     << count << " listings: \n";
59
```



```
60     for(int i=0; i< count ; ++i)
61     {
62         myList[i].ShowListing();
63     }
64
65     input.close();
66     cout << "\n\n And a gracious goodbye! \n";
67     return 0;
68 }
25 }
```

Output

In the PhoneList constructor.

Your Address Book contains these 5 listings:

Dr. Frank N. Stein	Castle, The Old Country	555-111-9999
Count Dracula	1 Red Ave. Bloodsworth	555-111-8888
Little Miss Muffet	29 Tuffet St, Arachnoid	555-111-7777
Mr. Creature	Black Lagoon North	555-111-6666
Fred WereWolf	London, England	555-111-5555

And a gracious goodbye!

The PhoneList.cpp file contains only two class function definitions. The *set* functions are all contained in the class declaration.

Program 7-7

```
1 //File: PhoneList.cpp
2
3 #include "PhoneList.h"
4 #include <string>
5 #include <iostream>
6 #include <iomanip>
7
8 using namespace std;
9
10 PhoneList::PhoneList()
11 {
12     // constructor sets strings to ""
13     // and writes a message
14     name = "";
15     address = "";
16     phNumber = "";
17     cout << "\n In the PhoneList constructor.";
18 }
19
```

```

20 void PhoneList::ShowListing()
21 {
22     cout << endl << setw(20) << name << setw(25)
23         << address << setw(15) << phNumber;
24
25 }
```

7.6

Overloaded Operators and Objects

The C++ language provides many symbol operators. The C++ operators are designed for specific tasks and data types. There are relational (`<`, `>`, `<=`, `>=`, `==`, `!=`), logical (`&&`, `||`, `!`), and arithmetic (`+`, `-`, `*`, `/`, `%`) operators. Programmers find the increment and decrement (`++`, `--`) operators convenient to use when working with integers. The modulus (`%`) operates only on integers.

Is it possible to use operators with objects? Can we increment an object? Is it possible to know if one object is greater than another? Can we add two objects together?

The good news is that there is a technique in C++ to overload operators. Just as we overloaded functions, we can also overload operators. Recall that when we overloaded a function, we had more than one implementation but only one interface—that is, we had several functions with the same name, such as *DrawLine* and *SayGoodnight*.

We already have some ***overloaded operators*** in C++. An asterisk, `*`, tells the program to perform multiplication when the asterisk is used with two numbers. When the asterisk is paired with a pointer, the asterisk is the indirection operator and accesses the memory to which the pointer is pointing.

It is possible to overload operators in C++. C++ operators (which are just symbols) perform different tasks, depending on how they are used. To overload an operator, we must write a specific operator function that is a class member. This ***operator function*** defines a valid C++ operator that performs the tasks when it is working with an object of that class. You may use only valid C++ operators. *Note:* You cannot use characters that are not already operators in C++. And unary operators must be overloaded as unary.

The prototype format for an overloaded operator function contains the keyword ***operator*** and is followed by the desired operator symbol:

```
return_type operator symbol (input parameter list);
```

It is easier to see overloaded operations in action, so our next program has a *BaseballPlayer* class. We'll compare baseball player objects.

Binary and Unary Operators In Classes

The *BaseballPlayer* class represents a single baseball player. It includes data for his name, number of home runs, and number of runs batted in (RBIs). There are *set* functions for the data, we've overloaded the greater than (`>`), less than (`<`) and exclamation point (`!`) operators for this class. When you overload your operators you have to determine what characteristics makes one object “greater than” another

overloaded operator

a valid C++ symbol (operator) that has been assigned a specialized task when the operator is used with objects

operator function

a class member function that is required to specify the task(s) for overloading an operator

operator

C++ keyword for designating class member operator functions

object. In this case, when we compare two baseball players, let's say that the player who has more home runs is “greater.” If the players have the same number of home runs, then the player with more RBIs is “greater.” For convenience, let's overload the `!` operator so that it increments the number of home runs.

Before we look at the details in the class code, let's see how easy it is to use overloaded operators in a class. In our `main` function we create two baseball players, Aaron and Jorge. We use the overloaded constructor to pass in their data when the objects are created. There is a function called `CheckWhoIsGreater` that is passed the baseball player objects references. In this function we compare who is greater, and write the results to the screen. We also make use of the `!` operator to increment Jorge's home runs.

Program 7-8

```
1 //File: Driver.cpp
2
3 #include <iostream>
4 #include <string>
5 #include "BaseballPlayer.h"
6 using namespace std;
7
8 void CheckWhoIsGreater(BaseballPlayer &p1,
9                         BaseballPlayer &p2);
10 int main()
11 {
12     cout << "\n The Baseball Players Program! ";
13
14     //Aaron is our first player object
15     //He has 25 homers and 28 RBIs
16     BaseballPlayer bb1("Aaron", 25, 28);
17
18     //Jorge is our second player object
19     //He has 24 homers and 26 RBIs
20     BaseballPlayer bb2("Jorge", 24, 26);
21
22     cout << " Here are our two players: \n";
23     bb1.WritePlayerStats();
24     bb2.WritePlayerStats();
25
26     CheckWhoIsGreater(bb1, bb2);
27
28     //Jorge is going to hit a home run!
29     !bb2;
30
31     CheckWhoIsGreater(bb1, bb2);
32
33     //Jorge hits another one!
34     !bb2;
35
36     CheckWhoIsGreater(bb1, bb2);
```





```
37
38     return 0;
39 }
40
41 void CheckWhoIsGreater(BaseballPlayer &p1,
42                         BaseballPlayer &p2)
43 {
44     cout << "\n Checking: Who is \"greater\"?";
45     if(p1 > p2)
46     {
47         cout << "\n " << p1.GetName() <<
48             " is greater!";
49         p1.WritePlayerStats();
50     }
51     else
52     {
53         cout << "\n " << p2.GetName() <<
54             " is greater!";
55         p2.WritePlayerStats();
56     }
57 }
```

Output

The Baseball Players Program! Here are our two players:
Aaron has 25 home runs and 28 RBIs
Jorge has 24 home runs and 26 RBIs

Checking: Who is "greater"?
Aaron is greater!
Aaron has 25 home runs and 28 RBIs

Jorge just hit a homer!
He now has 25 home runs!

Checking: Who is "greater"?
Aaron is greater!
Aaron has 25 home runs and 28 RBIs

Jorge just hit a homer!
He now has 26 home runs!

Checking: Who is "greater"?
Jorge is greater!
Jorge has 26 home runs and 26 RBIs.

In the `BaseballPlayer` class, we designate the operator in the function prototype. Here we are saying that the `>`, `<`, and `!` operators have special meaning for this class, and when we write code such as:

```
if(p1 > p2)      //returns a true if p1 is "greater"
```

or

```
!p2;           //increments the object's number of home runs
```

Examine the BaseballPlayer class:

Program 7-8

```
1 //File: BaseballPlayer.h
2
3 #ifndef _BBP_H
4 #define _BBP_H
5
6 #include <string>
7 using namespace std;
8
9 class BaseballPlayer
10 {
11     private:
12         string name;
13         int homers, RBI;
14
15     public:
16         BaseballPlayer();
17         BaseballPlayer(string n, int h, int rbi);
18
19         void setName(string n){name = n;}
20         void setHomers(int num){homers = num;}
21         void setRBIs(int rbi){RBI = rbi;}
22         void WritePlayerStats();
23
24         string GetName(){ return name; }
25
26         //The > operator "bigger" if homers is larger
27         //If same number of homers, greater RBIs is larger
28         bool operator > (BaseballPlayer bbp);
29         bool operator < (BaseballPlayer bbp);
30
31         //The ! operator increments the number of homers.
32         void operator !();
33     };
34
35 #endif
```

The fun begins in the BaseballPlayer.cpp file. The overloaded operators are just regular class functions and follow the same rules. First, look at this file, then we'll discuss the operators in detail.

```
Program 7-8
1 //File: BaseballPlayer.cpp
2
3 #include "BaseballPlayer.h"
4 #include <iostream>
5 #include <string>
6
7 using namespace std;
8
9 BaseballPlayer::BaseballPlayer()
10 {
11     name = "";
12     homers = RBI = 0;
13 }
14
15 BaseballPlayer::BaseballPlayer(string n,int h, int rbi)
16 {
17     name = n;
18     homers = h;
19     RBI = rbi;
20 }
21
22 bool BaseballPlayer::operator > (BaseballPlayer bbp)
23 {
24     if( homers > bbp.homers)
25         return true;
26     else if(homers == bbp.homers)
27     {
28         if(RBI > bbp.RBI)
29             return true;
30         else
31             return false;
32     }
33     else
34         return false;
35 }
36
37 bool BaseballPlayer::operator < (BaseballPlayer bbp)
38 {
39     if(homers < bbp.homers)
40         return false;
41     else if(homers == bbp.homers)
42     {
43         if(RBI < bbp.RBI)
44             return false;
45         else
46             return true;
47     }
```

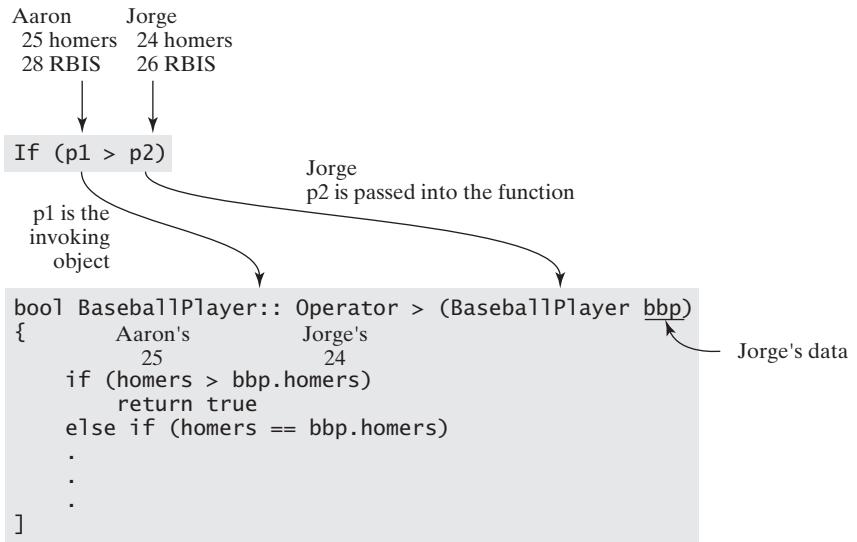
```
48     else
49         return true;
50 }
51
52 void BaseballPlayer::operator ! ()
53 {
54     ++homers;
55     cout << "\n " << name << " just hit a homer!";
56     cout << "\n He now has " << homers << " home runs!\n";
57 }
58
59 void BaseballPlayer::WritePlayerStats()
60 {
61     cout << "\n " << name << " has " << homers <<
62         " home runs " << "and " << RBI << " RBIs " << endl;
63 }
```

We have two types of operators in this code, binary and unary. First, let's look at the binary operators. The `>` and `<` operators require two operands. In other words, each of these operators are designed to compare two objects. Both objects “come equipped” with these operators, but you need to look at the comparison statement to realize that we are passing one object into the other object’s operator function. Inside this function we can write the code to determine which player is greater. Look at the `>` operator code here:

```
bool BaseballPlayer::operator > (BaseballPlayer bbp)
{
    if(homers > bbp.homers)
        return true;
    else if(homers == bbp.homers)
    {
        if(RBI > bbp.RBI)
            return true;
        else
            return false;
    }
    else
        return false;
}
```

The `bbp` object is passed into the operator function where comparison code is written. Notice that we use the class data “`homers`” and “`bbp.homers`.” The object on the left side of the operator is the ***invoking object***, meaning that this function will use that object’s data variables as the class variables. It is easier to see this in a diagram than it is to read about it in a paragraph. Please study Figure 7-10.

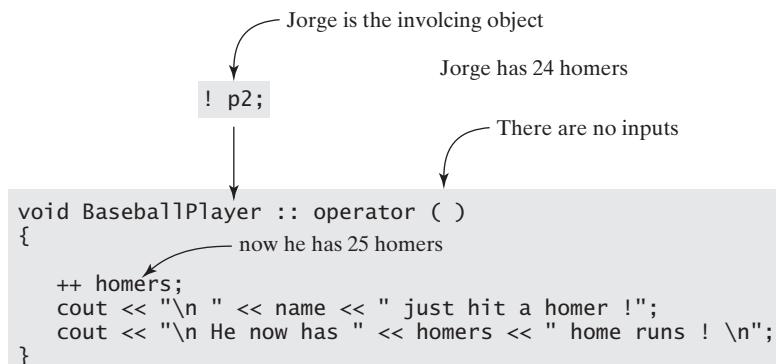
invoking object
the object who’s
data the class mem-
ber function is cur-
rently using

**Figure 7-10**

Binary operator `>` in the `BaseballPlayer` class.

A unary operator requires only one operand. We establish the `BaseballPlayer` class so that an integer represents the number of home runs the player has hit. The `!` operator now has a new definition when used with a `BaseballPlayer` object. Examine Figure 7-11.

Overloaded operators may be a bit advanced for the beginning C++ student, but it is a very useful tool, especially when using the Standard Template Library classes that sort their objects. There is another example in the Practice section of this text, as well as some problems in the back of the chapter. Don't be afraid to test your skill by overloading your C++ operators in your classes!

**Figure 7-11**

Unary Operator `!` increments the number of home runs for the invoking object in the `BaseballPlayer` class.

7.7

Pointers, References, and Classes

C++ allows an object's address to be passed to a function in the same fashion as we passed data variable addresses. The object's address may be explicitly passed to a pointer, or its address is passed implicitly into a reference variable. The function then accesses the class' public members. We see a different operator when the address of an object is passed to a pointer in a function. This new operator, the right-arrow operator (\rightarrow), is used to access the public members. We use the dot operator if the object's reference has been passed.

Pointers and References Program Example

In Program 7-9, we compare how the addresses of two Kitty objects are passed to functions, and how the public methods are accessed. This is a silly program on the outside, but very interesting on the inside. This example contains a Kitty class and four separate (non-Kitty class) functions. We have overloaded the *FillCatInfo* and *PrintCatInfo* functions so that each version uses a pointer to a Kitty, as well as a reference to a Kitty. The *FillCatInfo* fills the Kitty object data by calling the *set* functions. To keep it simple, we've hard coded the data in these functions. Spend time examining Program 7-9 and learning the mechanics of passing an address of an object to a function.

First, we declare the Kitty class. We need three *set* functions to place information into the private data variables and a *print* function.

Program 7-9

```
1 //File: Kitty.h
2
3 #ifndef _KITTY_H
4 #define _KITTY_H
5
6 #include <iostream>
7 #include <string>
8 using namespace std;
9
10 class Kitty
11 {
12     private:
13         string breed, name;
14         string favFood;
15
16     public:
17         Kitty();
18         void SetBreed(string b){breed = b;}
19         void SetName(string n) {name = n; }
20         void SetFavFood(string ff) {favFood = ff; }
21         void PrintCatInfo();
22 };
```

```
23  
24 #endif
```

The Kitty class functions are easy to write. The *set* functions are in the *Kitty.h* and class declaration. The constructor and *Print* functions are located in *Kitty.cpp*.

Program 7-9

```
1 //File: Kitty.cpp  
2  
3 #include "Kitty.h"  
4 #include <iostream>  
5 #include <string>  
6  
7 using namespace std;  
8  
9 Kitty::Kitty()    //constructor  
10 {  
11     breed = "unknown";  
12     name = "Kitty Kitty";  
13     favFood = "fresh chicken";  
14 }  
15  
16 void Kitty::PrintCatInfo()  
17 {  
18     cout << "\n " << name << " is a(n) " << breed  
19         << " cat who loves " << favFood;  
20 }
```

The *KittyDriver.cpp* contains the main, *FillCatInfo* and *PrintCatInfo* functions. Notice that the input arguments for the functions are a pointer and a reference to a Kitty.

Program 7-9

```
1 //File: KittyKitty.cpp  
2  
3 //This program demonstrates how to use  
4 //pointers and references with objects.  
5  
6 #include "Kitty.h"  
7  
8 //prototypes, one with a pointer,  
9 //one with a reference  
10 void FillCatInfo(Kitty *pCat);  
11 void FillCatInfo(Kitty &rCat);  
12  
13 void PrintCatInfo(Kitty *pCat);  
14 void PrintCatInfo(Kitty &rCat);  
15
```

```
16 int main()
17 {
18     Kitty meow, yowl;
19
20     cout << "\n The Kitty-Cat Program \n";
21
22     //we pass the address to a pointer
23     FillCatInfo(&meow);
24
25     //we pass the reference here
26     FillCatInfo(yowl);
27
28     PrintCatInfo(&meow);           //pointer
29     PrintCatInfo(yowl);          //reference
30
31     cout << "\n\n No more kitties for you. \n ";
32     return 0;
33 }
34
35 void FillCatInfo(Kitty * pCat) //pointer use ->
36 {
37     pCat->SetBreed("Alley Cat");
38     pCat->SetName("Thomas" );
39     pCat->SetFavFood("Milk");
40 }
41
42 void FillCatInfo(Kitty & rCat) //reference use .
43 {
44     rCat.SetBreed("Siamese" );
45     rCat.SetName("Comehere Kitty");
46     rCat.SetFavFood("Fresh birdies");
47 }
48
49 void PrintCatInfo(Kitty *pCat)
50 {
51     pCat->PrintCatInfo();
52 }
53
54 void PrintCatInfo(Kitty &rCat)
55 {
56     rCat.PrintCatInfo();
57 }
```



Output

The Kitty-Cat Program

Thomas is a(n) Alley Cat cat who loves Milk

Comehere Kitty is a(n) Siamese cat who loves Fresh birdies

No more kitties for you..

Date Time Demo Program

While we are looking at programs that use pointers and the right-arrow operator, now is a good time to see a short program that obtains the system time and date. We used this code earlier in the Date programs, but here is a complete demo for time and date. See Program 7-10. If your program needs to write the time and/or date, it is obtained easily by using the `_strftime` and `_strdate` functions provided in the `ctime` library. Simply make character arrays and pass them to the functions.

If you need the integer values for the date and/or time we need to use data types and functions that were originally defined in the C `time.h` library. The code may look strange here, but bear with us. We declare a variable of the type `time_t` (here, we call it *SystemTime*). The `time()` function fills this data variable with the number of seconds since 1/1/70, known as the Coordinated Universal Time (UTC). It was previously referred to as Greenwich Mean Time (GMT). We convert this time value to a `struct tm` by calling the `localtime` function found in the time library. Once we have filled the `OStime` structure, we can access the hours, minutes, seconds, month, day, year (and other values).

Program 7-10

```
1 //File: DateTimeDemo.cpp
2 //This program shows how to access the
3 //computer's system time and date.
4
5 //We use the C language time and date functions.
6
7 #include <iostream>
8 #include <ctime>
9 using namespace std;
10
11 int main()
12 {
13     cout << "\n What is the date and time? \n\n";
14
15     //If you just need a char array for the
16     //date and time, can use _strftime and _strdate.
17
18     char timeRightNow[20];
19     char dateRightNow[20];
20
21     // The _strftime function fills a character
22     // array with the system time.
23     // The _strdate function gives us the date.
24
25     _strftime(timeRightNow);
26     _strdate(dateRightNow);
27     cout << "\n From the _strdate and _strftime functions:";
```

```
28     cout << "\n The date is " << dateRightNow
29     << "\n The system time is " << timeRightNow ;
30
31 // If we need integer values for hr, min, sec
32 // or month, day year, we use a time_t struct
33 // that is defined in time.h
34
35 // UTC time format variable
36 time_t SystemTime;
37
38
39 // Pass the address of SystemTime to time()
40 // It fills the SystemTime with UTC seconds
41
42 time(&SystemTime);
43
44
45 // Need to convert UTC to something we can use.
46 // The localtime function does that.
47 // Declare tm ptr to hold individual time info
48
49 struct tm *OStime;
50
51 // Pass the address of SystemTime to localtime()
52 // It converts to tm struct, which has all we need.
53
54 OStime = localtime(&SystemTime);
55
56 // Now we access the hr, min, sec of system time,
57 // by using the -> operator
58
59 int hour = OStime->tm_hour;
60 int min = OStime->tm_min;
61 int sec = OStime->tm_sec;
62
63 int month = OStime->tm_mon;      //0 = Jan, 11 = Dec
64 int day = OStime->tm_mday;
65 int year = OStime->tm_year;      //based from 1900
66
67 cout << "\n\n From the data struct:";
68 cout << "\n The date is " << month + 1
69     << "/" << day << "/" << year + 1900;
70 cout << "\n The time is " << hour << ":"
71     << min << ":" << sec << endl;
72
73 return 0;
74 }
```

Output

What is the date and time?

From the `_strdate` and `_strftime` functions:

The date is 09/29/06

The system time is 19:32:33

From the data struct:

The date is 9/29/2006

The time is 19:32:33

7.8 Summary

In object-oriented programming, the design of the software is based on real-world objects instead of data flow. It is possible to incorporate the data and functions together in a class declaration. That accomplished, the programmer may create and use these class objects. The class should be thought of as a “job description” with the object performing the tasks defined in the description.

There are several software development principles in object-oriented programming. One principle is **encapsulation**, which means that the class data is “hidden” (i.e., placed in the private section of the class declaration). This step ensures that only class member functions can use the data and that the object data are not accessed by parts of the program that have no business tinkering with them. Table 7-1 summarizes the principles required of an object-oriented language.

As with any field of study, there is new terminology used to describe it. We have introduced several of these terms in this chapter. Table 7-2 provides a list and description of commonly used object-oriented terminology.

Class constructors, which may be overloaded, are executed automatically when an object is declared. The purpose of the constructor is to initialize the object data (or states) to known values. Destructor functions are executed when the object is destroyed or the program is terminated. Overloaded operator functions allow any C++ operators to be defined tasks with objects.

An Overview of Common Errors while Writing Classes



Beginning object-oriented programmers make a few common mistakes as they start to write classes and objects. These errors are more annoying than life-threatening because the compiler or linker often catches them, but they result in compiler or linker reports that may be misleading to the new programmer. Once you begin writing your own classes, you are bound to see a few of these errors. We cover the common ones here.

Error C2065: ‘description’ : Undeclared Identifier

The “undeclared identifier” occurs regularly if the programmer forgets the class name and scope operator. The programmer should separate his code into different files by placing the class declaration in the *.h file and the class definition in the *.cpp file. The programmer often forgets the class name and scoping operator

TABLE 7-1

Principles Required of an Object-Oriented Language

Principle	Description	Notes
Modularity	Creates logically separate units (objects) that are well defined and interact through the use of functions. Ideally, objects should have minimum dependence on each other.	Extends the idea of a function or subroutine so that data are included.
Encapsulation	The ability to package or wrap things together into a well-defined unit or class.	The encapsulation concept also allows the internal architecture and data to be hidden from other classes.
Abstraction	The ability to hide the complexity of the design and processes that carry out the operations on an object.	The abstraction concept deals with hiding how the functions work on the data. A class function is like a black box that performs its job without letting the world “see” how it does it.
Polymorphism	One interface, many methods (or implementations).	C++ implements polymorphism with inherited classes. (See Chapter 8.)
Inheritance	The ability to create new (derived) classes from existing (base) classes. Inheritance represents an “is a” class relationship.	Classes form a hierarchical relationship.
Persistence ^a	In distributed or database applications, the object states are retained (persist) after the program has been terminated.	C++ can mimic persistence using files or databases to store object states.
Concurrency ^a	In client/server applications, the objects are distributed across different hardware platforms.	A banking system mainframe computer and ATM machines are distributed across different platforms. The objects need to interact with each other through the use of well-defined interfaces (functions).

^aOptional object-oriented language requirement.

in the function header line. The Date class’ *GetFormattedDate* below illustrates this mistake. Because this looks like a standalone function, the class variables appear to be undeclared. The compiler reports that all four class members are undeclared (description, month, day, and year).

```
string GetFormattedDate()           //<== WHOOPS FORGOT THE Date:::  
{  
    stringstream strDate;  
    strDate << description;  
  
    string monName[12] = {"January", "February ", "March",  
    "April", "May", "June", "July", "August",  
    "September", "October", "November", "December"};
```

TABLE 7-2

Object-Oriented Terminology

Term	Description	Notes
Class	The data variables and functions that describe the job of a program component.	The format of a class contains data and function prototypes. The keyword “class” is used and is considered a data type in C++. A class sets up a blueprint for an object. Think of a class as a job description.
Object	An instance of a class. It is not “real” until the program is executed.	The object is the “worker” who performs the job description defined in the class.
Method	An Ada ^a language term for a function. It is often used in object-oriented languages to describe a function that belongs to a class.	In C++, functions can be referred to as methods or operations. Class functions are sometimes referred to as member functions.
Member or class member	Either a data variable or function that belongs to a class.	Both the data items and functions are referred to as class members
Object-oriented analysis (OOA)	When determining program requirements, the programmer must identify the necessary objects along with their characteristics and functions.	This task is usually accomplished with pencil and paper. Redundant characteristics are often discovered.
Object-oriented design (OOD)	Once the objects for a program are identified, the required functions and characteristics are refined and the interfaces to the objects are identified.	Object-oriented principles should be considered when designing the class.

^aThe Ada programming language was developed in the 1970s and named after Lady Ada Lovelace, daughter of poet Lord Byron.

```

        strDate << ":" << monName[month-1] << " " << day
        << ", " << year;
        return strDate.str();
    }
}

```

unresolved external symbol “private: void __thiscall Date:: DetermineLeapYear(void)”

Ah, the old “unresolved external symbol” error. It is generated by the linker as it tried to build the executable. If the linker has seen the prototype of a class function, but does not find the class function itself, it generates this type of error. In our Date class, we declare the void *DetermineLeapYear()* in the class declaration. If this function is not defined, i.e., you haven’t written it yet, you’ll see this type of error.

Building your program in stages is exactly the correct way to develop software. Anytime you have written the function in a class, you need to write the associated function too—else the linker complains. Don’t forget that you can write an

empty function to get going, and go back then fill in the code later. Writing your code this way will keep the linker happy. (You can write empty functions using a set of braces beside the prototype, such as `void DetermineLeapYear() {}`. Just don't forget to remove the `{}` here when you write the actual function.)

Error C2511: 'SetDate' : overloaded member function 'void (int,int,int)' not found in 'Date'

Overloaded functions are functions with the same name but have different input lists. The data type and variables in the function prototype and function header line must match, or the programmer is guaranteed problems. In our previous chapters, these types of errors were reported as type mismatches. When programming classes, the programmer will find a “new” type of error for this “old” problem.

Using the Date class again, we illustrate this error by writing the `SetDate` prototype correctly with four inputs, while the function header line only has three. Here is the prototype in the class declaration:

```
void SetDate(int m, int d, int y, string desc);           //4 inputs
```

The compiler is not happy with the code because it is expecting to find the form described in the prototype, but it sees the form below and reports that it cannot find an overloaded function. The `SetDate` function that generates the above error is shown below:

```
void Date::SetDate(int m, int d, int y)    <== WHOOPS, forgot the 4th input
{
    month = m;
    day = d;
    year = y;
    description = desc;
    DetermineLeapYear();
    CalcDayOfYear();
}
```

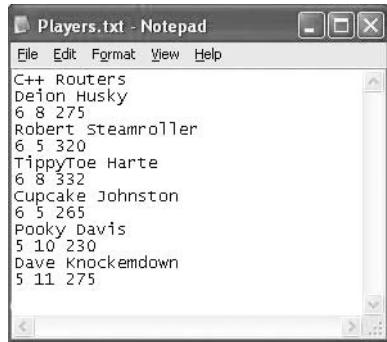
7.9

Practice!

Big, Bigger, Biggest Football Players

Our first practice program in this chapter has something for everyone! Program 7-11 has a `FootballPlayer` class that contains data and functions for a football player. The player information is read into an array of `FBPlayer` objects from a data file called `Players.txt`. The program first reports the player information as read from the file, sorts the players (from smallest to largest), and then reports the sorted information. The data in the `Players.txt` file includes the team name, each player's name and stats, including his weight and height in feet and inches. Figure 7-12 shows the file in Notepad.

The overloaded greater than operator, `>`, in the `FootballPlayer` class enables us to write a function to compare two players. A modified `Bubble Sort` function is used

**Figure 7-12**

The Players.txt data file.

to sort FootballPlayers. One player is greater than another player if he is taller; if the two are the same height, the heavier player is deemed greater. The program writes out the sorted players and reports the players in order from smallest to largest.

Here we declare the FootballPlayer (FBPlayer) class in the FBPlayer.h file. The associated class functions are in FBPlayer.cpp:

Program 7-11

```
1 //File: FBPlayer.h
2
3 #ifndef _FBPLAYER_H
4 #define _FBPLAYER_H
5
6 #include <string>
7 #include <iostream>
8 using namespace std;
9
10 class FBPlayer
11 {
12 private:
13     int weight, feet, inches, totalinches;
14     string name;
15 public:
16     FBPlayer();
17     void SetName(string n) { name = n;}
18     void SetWeight(int w) { weight = w;}
19     void SetHeight(int ft, int in);
20     void WriteInfo();
21
22     //the taller player is "bigger"
23     //if they are the same height, then
24     //the heavier player is "bigger"
25     bool operator > (FBPlayer p);
26 };
27
28 #endif
```

Program 7-11

```
1 //File: FBPlayer.cpp
2
3 #include "FBPlayer.h"
4 #include <iostream>
5 #include <iomanip>
6 #include <string>
7 using namespace std;
8
9 FBPlayer::FBPlayer()
10 {
11     weight = 0;
12     feet = inches = totalinches = 0;
13     name = "";
14 }
15
16 bool FBPlayer::operator > (FBPlayer p)
17 {
18     //the taller player is "bigger"
19     if(feet > p.feet) return true;
20
21     //if two players are the same height,
22     //the heavier player is "bigger"
23     if(feet == p.feet)
24     {
25         if(inches > p.inches) return true;
26         else if(inches == p.inches)
27         {
28             if(weight > p.weight) return true;
29             else return false;
30         }
31         else return false;
32     }
33     else
34     return false;
35 }
36
37 void FBPlayer::SetHeight(int ft, int in)
38 {
39     feet = ft;
40     inches = in;
41 }
42
43 void FBPlayer::WriteInfo()
44 {
45     cout << endl << setw(25) << name
46     << setw(8) << feet << "' "
47     << setw(3) << inches << "\\" "
```

```
48         << setw(8) << weight << " pounds";
49 }
```

The FBSortDriver.cpp file contains the *main* function along with the *Read* and *Sort* functions. The *Read* function opens and read the file, obtaining the team name, then fills each FBPlayer object. It counts the players as it reads the data. The player data is written to the screen using a *for* loop. With each pass in the loop the individual player is asked to write its own data to the screen. Pretty neat, huh?

Program 7-11

```
1 //File: FBSortDriver.cpp
2 //This program demonstrates how overloaded
3 //operators in classes can be used to sort
4 //an array of Football Player objects.
5
6 #include "FBPlayer.h"
7 #include <iostream>
8 #include <fstream>
9 using namespace std;
10
11 #define FILE "Players.txt"
12
13 bool ReadFootballPlayers(string &rName,
14                           FBPlayer team[], int& rTotal);
15
16 void SortPlayers(FBPlayer team[], int total);
17
18 int main()
19 {
20     FBPlayer team[6]; //football player objects
21     string teamName;
22     int total;
23
24     cout << " Welcome to the Sorting Football "
25         " Players Program \n";
26
27     bool bError = ReadFootballPlayers(teamName, team, total);
28     if(bError)
29     {
30         cout << "\n\n couldn't find file " << FILE;
31         exit(1);
32     }
33
34     cout << "\n Team: " << teamName << endl;
35
36     cout << "\n The original order of players: ";
37     for (int i= 0; i < total; ++ i)
```

```
38     {
39         team[i].WriteInfo();
40     }
41
42     SortPlayers(team, total);
43
44     cout << "\n\n The sorted Players are: ";
45
46     for ( i= 0; i < total; ++ i)
47     {
48         team[i].WriteInfo();
49     }
50
51     cout << "\n\n These are big fellows! \n\n";
52
53     return 0;
54 }
55
56 void SortPlayers(FBPlayer team[], int total)
57 {
58     // a classic bubble sort for FBPlayers
59     // sorts from low to high
60
61     int i, j;
62     FBPlayer temp;
63     for(i = 0; i < (total - 1) ; ++i)
64     {
65         for(j = 1; j < total; ++j)
66         {
67             //this is possible since we've
68             //defined the > for FBPlayer objectsRe
69             if(team[j-1] > team[j] )
70             {
71                 temp = team[j];
72                 team[j] = team[j-1];
73                 team[j-1] = temp;
74             }
75         }
76     }
77 }
78
79 bool ReadFootballPlayers(string &rTeamName,
80                         FBPlayer team[], int &rTotal)
81 {
82     //opens and reads the input filename,
83     //and fills the team data
84
85     int wt, ft, in;
```



```
86     string buffer;
87
88     ifstream input;
89     input.open(FILE);
90
91     if(!input) return true;
92
93     //first read the team name
94     getline(input,rTeamName);
95
96     int i = 0;
97
98     while (! input.eof() )
99     {
100        //Read the name
101        getline(input, buffer);
102        team[i].SetName(buffer);
103
104        //now read 3 ints
105        input >> ft >> in >> wt;
106
107        //strip off end of line
108        input.ignore();
109
110        //set data into object
111        team[i].SetWeight(wt);
112        team[i].SetHeight(ft,in);
113
114        ++i;
115    }
116
117    rTotal = i;
118    input.close();
119
120    return false;
121 }
```

Output

Welcome to the Sorting Football Players Program

Team: C++ Routers

The original order of players:

Deion Husky	6' 8"	275 pounds
Robert Steamroller	6' 5"	320 pounds
TippyToe Harte	6' 8"	332 pounds
Cupcake Johnston	6' 5"	265 pounds
Pooky Davis	5' 10"	230 pounds
Dave Knockemdown	5' 11"	275 pounds

The sorted Players are:

Pooky Davis	5' 10"	230	pounds
Dave Knockemdown	5' 11"	275	pounds
Cupcake Johnston	6' 5"	265	pounds
Robert Steamroller	6' 5"	320	pounds
Deion Husky	6' 8"	275	pounds
TippyToe Harte	6' 8"	332	pounds

These are big fellows!

The Teacher's Helper, The Grader Program

The Teacher's Helper program performs the arduous task of computing students' grades and writing the final grade information to a data file. The college, course, student names, IDs, and scores for four-hour exams are shown in the Student-Grades.txt file as seen in Figure 7-13.

The program is designed using two classes, a Student class, and a Grader class. The Student class contains data for the student including name, identification number, an integer array for test scores, and a float for test average. Appropriate *set* and *get* functions are included in the class. Notice how we compute the test average when the four test scores are set into the class. Here are the Student class files:

```
Program 7-12
1 //File: Student.h
2
3 #ifndef _STUDENT_H
4 #define _STUDENT_H
5
6 #include <string>
7 using namespace std;
8
9 class Student
```

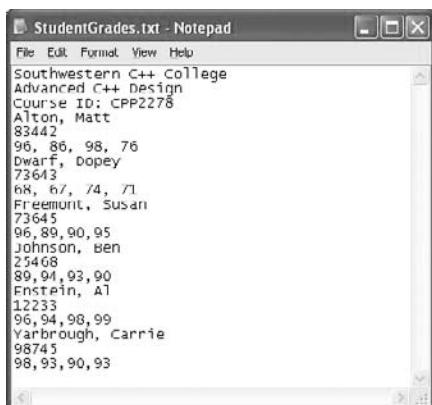


Figure 7.13
The StudentGrades.txt data
file that is used in the Teacher's
Helper program.

```
10  {
11  private:
12      string name, ID;
13      int testScore[4];
14      float ave;
15
16  public:
17      Student();
18
19      void SetNameID(string n, string i);
20      void SetTestScores(int sc[]);
21
22      float GetAverage(){ return ave; }
23      string GetName(){ return name; }
24      string GetID(){ return ID; }
25  };
26
27 #endif
28 }
```

Program 7-12

```
1 //File: Student.cpp
2
3 #include "Student.h"
4
5 Student::Student()
6 {
7     name = ID = "";
8     ave = static_cast<float>(0.0);
9
10    for(int i = 0; i < 4; ++i)
11        testScore[i] = 0;
12 }
13
14 void Student::SetNameID(string n, string i)
15 {
16     name = n;
17     ID = i;
18 }
19
20 void Student::SetTestScores(int sc[])
21 {
22     int i;
23
24     //copy the test scores into the array
25     for(i = 0; i < 4; ++i)
26         testScore[i] = sc[i];
```

```
27     float sum = 0;
28
29     //sum and determine average
30     for(i = 0; i < 4; ++i)
31         sum += testScore[i];
32
33     ave = sum/4.0;
34 }


---


```

The Grader class contains an array of Students. C++ string objects are used for the course information. There is a private *Read* function that is called from the constructor. The *Compute* and *Write* functions are public, as we expect our user to specifically call them. The *ComputeCourseStats* function has an easy job of calculating the course data. Each Student object computes its own average. The Grader merely loops through the Student objects, asking for the average value. The high and low students are determined as well. Techniques are described through comments in the source code.

Note! You might be tempted to have the constructor run the entire program. Do not fall into this poor-programming style! Remember, the constructor's job is to initialize class data! We can have the constructor read the data because it fills the class data in preparation of the actual work. But further work by the Grader should be initiated via calls by the *Grader* object.

```
Program 7-12
1 //File: Grader.h
2
3 #ifndef _GRADER_H
4 #define _GRADER_H
5
6 #include "Student.h"
7 #include <string>
8 using namespace std;
9
10 class Grader
11 {
12     private:
13         Student kids[10];
14         int total;
15         string courseName, courseID, college;
16         float courseAve, courseHigh, courseLow;
17         int highKid, lowKid;
18         void ReadStudentFile();
19
20     public:
21         Grader();
22         void ComputeCourseStats();
```

```
23     void WriteResultsFile();
24 }
25
26 #endif
```

Program 7-12

```
1 //File: Grader.cpp
2
3 #include "Grader.h"
4
5 #include <fstream>
6 #include <iostream>
7 #include <iomanip>
8 using namespace std;
9
10 #define FILE "StudentGrades.txt"
11
12 Grader::Grader()
13 {
14     total = 0;
15     ReadStudentFile();
16 }
17
18 void Grader::ComputeCourseStats()
19 {
20     //loop through array of kids
21     //obtaining the students' average
22     //to compute course average
23     //remember high and low ave too
24
25     float sum = 0;
26     courseHigh = kids[0].GetAverage();
27     courseLow = courseHigh;
28     lowKid = highKid = 0;
29
30     float score;
31
32     for(int i = 0; i < total; ++ i)
33     {
34         score = kids[i].GetAverage();
35         sum = sum + score;
36
37         if(score < courseLow)
38         {
39             lowKid = i;
40             courseLow = score;
41         }
42     }
43 }
```

```
42     if(score > courseHigh)
43     {
44         highKid = i;
45         courseHigh = score;
46     }
47 }
48
49 courseAve = sum/total;
50 }
51
52
53 void Grader::WriteResultsFile()
54 {
55     ofstream output;
56
57     string file;
58     cout << "\n Please enter the output file ";
59     getline(cin,file);
60
61     //convert the string to a c-string
62     output.open(file.c_str());
63
64     if(!output)
65     {
66         cout << "\n Trouble opening output file."
67             << "\n Can't write the file. " << endl;
68     }
69
70     //set output's precision
71     output.setf(ios::fixed);
72     output.precision(2);
73
74     output << "\n Grade Results for " << college
75             << "'s \n " << courseName
76             << "\n ID " << courseID << endl;
77
78     string stName, stID;
79     float stAve;
80
81     output << setw(20) << "Name" << setw(8)
82             << "ID" << setw(10) << "Ave" << endl;
83
84     for(int i = 0; i < total; ++i)
85     {
86         stName = kids[i].GetName();
87         stID = kids[i].GetID();
88         stAve = kids[i].GetAverage();
89
90         output << setw(20) << stName << setw(8)
```

```
91             << stID << setw(10) << stAve << endl;
92         }
93
94         output << "\n Course Ave " << courseAve
95             << "\n High Ave was " << courseHigh
96             << " by " << kids[highKid].GetName()
97             << "\n Low Ave was " << courseLow
98             << " by " << kids[lowKid].GetName();
99
100        output.close();
101        cout << "\n All Done! \n Check " << file
102            << " for " << courseName << " grade results!"
103            << endl;
104    }
105
106 void Grader::ReadStudentFile()
107 {
108     ifstream input;
109     input.open(FILE);
110     if(!input)
111     {
112         cout << "\n Can't find the "
113             << " StudentGrades.txt file."
114             << "\n Exiting program! ";
115
116     //hold for an Enter key, so the user sees the message
117     cin.get();
118     exit(1);
119 }
120
121 //Ok, open and ready to go
122 //read first line, college
123 getline(input,college);
124
125 //next two lines are the course and course ID
126 getline(input,courseName);
127 getline(input,courseID);
128
129 //rest of file are students, 3 lines per
130 //we'll read until the end of file
131 //counting students as we go
132
133 string name, id;
134 int g[4];
135 int count = 0;
136 char comma;
137
138 //each pass, read each student's data
```

```

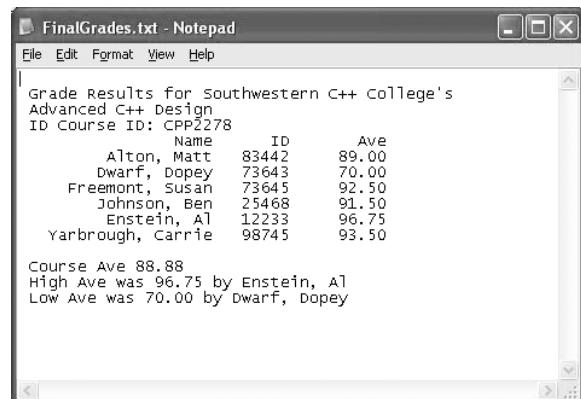
139     while(!input.eof() )
140     {
141         getline(input,name);
142         getline(input,id);
143         input >> g[0] >> comma >> g[1] >> comma
144             >> g[2] >> comma >> g[3];
145         input.ignore();
146
147         kids[count].SetNameID(name,id);
148         kids[count].SetTestScores(g);
149         ++count;
150     }
151
152 //set the total number of students in file
153 total = count;
154
155 cout << "\n We have read " << total
156     << " student grades from " << FILE << endl;
157 //close the file
158 input.close();
159 }
```

The GraderDriver.cpp file contains the *main* function. This function is short and sweet. We have a few hundred lines of code behind us now in this program. All the “heavy lifting” is performed by our classes. Isn’t that great! Sit back and enjoy our four-line main function. Figure 7-14 shows the output data file from this program.

Program 7-12

```

1 //File: GraderDriver.cpp
2 //This program uses the Grader class to
3 //help us determine grades for C++ students.
4
```



FinalGrades.txt - Notepad

Grade Results for southwestern C++ College's
Advanced C++ Design
ID Course ID: CPP2278

Name	ID	Ave
Alton, Matt	83442	89.00
Dwarf, Dopey	73643	70.00
Freemont, Susan	73645	92.50
Johnson, Ben	25468	91.50
Enstein, Al	12233	96.75
Yarbrough, Carrie	98745	93.50

Course Ave 88.88
High Ave was 96.75 by Enstein, Al
Low Ave was 70.00 by Dwarf, Dopey

Figure 7.14

The FinalGrades.txt output file from the teacher’s helper program.

```

5 #include "Grader.h"
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     cout << "\n The Teacher's Helper Program! \n";
12
13     Grader kelly;
14
15     kelly.ComputeCourseStats();
16
17     kelly.WriteResultsFile();
18
19     return 0;
20 }

```

Output

The Teacher's Helper Program!

We have read 6 student grades from StudentGrades.txt

Please enter the output file FinalGrades.txt

All Done!

Check FinalGrades.txt for Advanced C++ Design grade results!

The Stock Tank Calculator

In Chapter 2 we wrote the Circular Stock Tank Program (Program 2-26 on page 88). In this program we asked the cowboy to enter the diameter of his circular stock tank (in inches). We calculated the total volume in gallons that it would hold, as well as a percentage of the total. We're now going to expand this calculation to build a StockTankCalculator class that works on three shapes of tanks.

If you search for stock tanks on the Web, you'll find that farm and ranch watering equipment is available in three distinct shapes: circular, "oval," and rectangular. Figure 7-15 shows these three shapes.

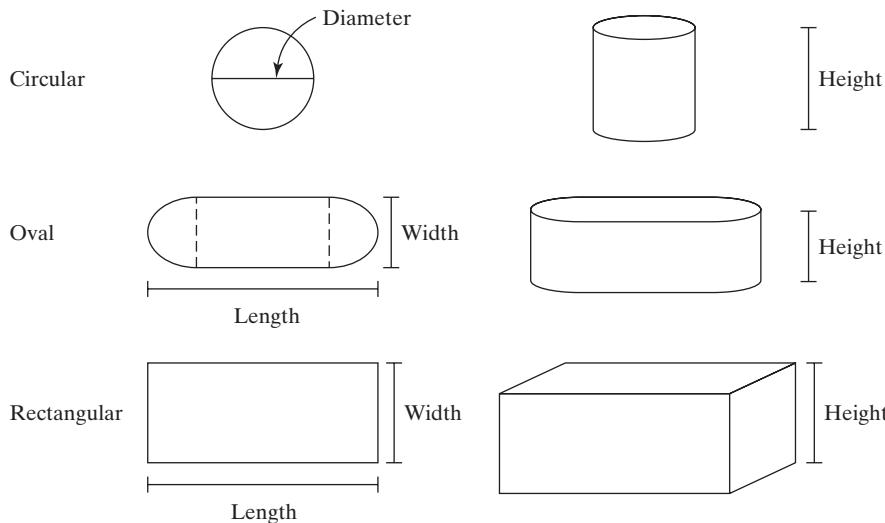
The Stock Tank Calculator uses the *set* functions for passing data into the class and *get* functions to retrieve data from the class. When the *set* function is called, the class assigns the data and then calls the private *CalcCapacity* function that determines the total capacity of the tank. Once this value is known, the percentage calculation is straightforward. As a default, we create the calculator object with a circular tank, 72" by 24". You see this initialization in the constructor. Examine the class files here. We use the cubic inch to gallon conversion factor again (231 cubic inches of water is one gallons).

Program 7-13

```

1 //File: StockTankCalc.h
2
3 #ifndef _STCalc_H

```

**Figure 7.15**

Three shapes that our StockTankCalculator can compute.

```
4 #define _STCalc_H
5
6 #include <string>
7 using namespace std;
8
9 class STCalc
10 {
11 private:
12     int tankL, tankW, tankH;
13     double tankGallons, tankPercent;
14     string tankShape;
15
16     void CalcCapacity();
17
18 public:
19     STCalc();
20
21     void setShape(string shape){tankShape = shape;}
22
23     void setCircTankDims(int height, int width);
24     void setRectTankDims(int height, int width, int length);
25     void setOvalTankDims(int height, int width, int length);
26
27     double getPercent(int per);
28     double getTotalCapacity();
29
```

```
30     void WriteTankData();  
31 };  
32  
33 #endif
```

Program 7-13

```
1 //StockTankCalc.cpp  
2 #include "StockTankCalc.h"  
3 #include <string>  
4 #include <iostream>  
5 #include <cmath>           //for pow function  
6  
7 using namespace std;  
8  
9 STCalc::STCalc()  
10 {  
11     //setting default tank to  
12     //Circular 72 x 24  
13     tankH = 24;  
14     tankW = 72;  
15     tankShape = "Circular";  
16     tankL = 0;  
17     CalcCapacity();  
18 }  
19  
20 void STCalc::setCircTankDims(int height, int width)  
21 {  
22     tankH = height;  
23     tankW = width;  
24 }  
25  
26 void STCalc::setRectTankDims(int height, int width,  
27                               int length)  
28 {  
29     tankH = height;  
30     tankW = width;  
31     tankL = length;  
32 }  
33  
34 void STCalc::setOvalTankDims(int height, int width,  
35                               int length)  
36 {  
37     tankH = height;  
38     tankW = width;  
39     tankL = length;  
40 }
```

```
41
42 double STCalc::getTotalCapacity()
43 {
44     CalcCapacity();
45     return tankGallons;
46 }
47
48 void STCalc::CalcCapacity()
49 {
50     double PI = 3.14159265;
51
52     if(tankShape == "Oval")
53     {
54         double endVol = (tankW/2) * (tankW/2)
55                         * PI*tankH;
56
57         double rectVol = (tankL-tankW) * tankW
58                         * tankH;
59
60         tankGallons = (endVol + rectVol)/231.0;
61     }
62     else if(tankShape == "Rectangular")
63     {
64         tankGallons = (tankL * tankW * tankH)/231.0;
65     }
66     else if(tankShape == "Circular")
67     {
68         double tankArea = pow( (tankW/2.0), 2)*PI;
69         tankGallons = tankArea*tankH/231.0;
70     }
71 }
72 }
73
74 double STCalc::getPercent(int percentage)
75 {
76     CalcCapacity();
77     tankPercent = tankGallons*percentage*.01;
78
79     return tankPercent;
80 }
81
82 void STCalc::WriteTankData()
83 {
84     CalcCapacity();
85
86     cout << " Shape: " << tankShape;
87     cout << " Total capacity: "
88         << tankGallons << " gallons";
```



```
89
90     cout << "\n Width: " << tankW
91     << "\n Height: " << tankH;
92
93     if(tankShape != "Circular")
94         cout<<"\n Length: "<<tankL;
95
96 }
```

The Driver.cpp file contains the *main* function. This *main* function is structured so that the programmer can test the various functions in the Stock Tank Calculator object. There is a *Menu* function and *switch* statement that provides us with the various options in program. Our output here is brief and it is probably easier for you to obtain these program files and run it yourself for full illustration. In our output we first ask the object to show us its default data, and indeed we see the default data established in the constructor.

Program 7-13

```
1 //StockTankDriver.cpp
2 #include <iostream>
3 #include <string>
4 #include "StockTankCalc.h"
5
6 using namespace std;
7
8 int Menu();
9
10 int main()
11 {
12     cout <<"\n The Stock Tank Calculator Program";
13
14     int choice, chooseShape;
15     int ht, diam, len, width;
16     int per;
17
18     double gallons;
19
20     STCalc Tank;
21
22     do
23     {
24         choice = Menu();
25
26         switch (choice)
27         {
28             case 1:           //choose shape of tank
29                 cout <<" Select stock tank shape\n";
```



```
30         cout <<" (1) Circular (2) Rectangular "
31                 <<"(3) Oval ";
32         cin >>chooseShape;
33
34         if(chooseShape == 1)
35     {
36             Tank.setShape("Circular");
37             cout <<"\nEnter diameter height: ";
38             cin >> diam >> ht;
39             Tank.setCircTankDims(diam, ht);
40
41         }
42         else if(chooseShape == 2)
43     {
44             Tank.setShape("Rectangular");
45             cout <<"\nEnter width height length: ";
46             cin >> width >> ht >> len;
47             Tank.setRectTankDims(width, ht, len);
48         }
49         else if(chooseShape == 3)
50     {
51             Tank.setShape("Oval");
52             cout <<"\nEnter width height length: ";
53             cin >> width >> ht >> len;
54             Tank.setOvalTankDims(ht, width, len);
55         }
56         else
57     {
58             cout <<"\n Whoops! Not a choice. \n";
59
60         }
61         break;
62
63     case 2:           //show tank data
64         Tank.WriteTankData();
65         break;
66
67     case 3:           //percentage of tank
68         cout << "\n\nEnter a percentage: ";
69         cin >> per;
70         gallons = Tank.getPercent(per);
71         cout << "\n When the tank is " << per
72             << "% full, it will hold "
73             << gallons << " gallons.";
74     }
75
76 }while(choice != 4);
77
```

```
78     return 0;
79 }
80
81 int Menu()
82 {
83     int choice;
84
85     cout <<"\n\n Stock Tank Calculator Menu\n";
86     cout << " (1) Set Tank Shape (3) Get % of capacity\n"
87         << " (2) See Tank Info (4) Exit\n\n";
88
89     cin >>choice;
90     cin.ignore();
91     return choice;
92 }
```

Output

The Stock Tank Calculator Program

Stock Tank Calculator Menu

- (1) Set Tank Shape (3) Get % of capacity
- (2) See Tank Info (4) Exit

2

Shape: Circular Total capacity: 423.013 gallons

Width: 72

Height: 24

Stock Tank Calculator Menu

- (1) Set Tank Shape (3) Get % of capacity
- (2) See Tank Info (4) Exit

3

Enter a percentage: 50

When the tank is 50% full, it will hold 211.507 gallons.

Stock Tank Calculator Menu

- (1) Set Tank Shape (3) Get % of capacity
- (2) See Tank Info (4) Exit

4

REVIEW QUESTIONS AND PROBLEMS

Short Answer

1. What is the purpose of the *Set* and *Get* functions in classes?
2. Who may access the private data members in a class?
3. What is the difference between a class declaration and class definition?
4. What are the steps a programmer must take when creating a class that will result in an array of objects?

5. When is the constructor function executed? What is its primary purpose?
6. Describe the “ideal” file organization scheme to use with classes.
7. Name one operator that is already overloaded in the C++ language—that is, depending on how it is used dictates the job that it performs.
8. How many destructors may a class contain? What is the main purpose of the destructor? When is this function called and who may call it?
9. Declaring a class with all the data and functions public violates which principle(s) of object-oriented programming?
10. Describe the difference in the call statement if a class’ public member function is called from outside the class versus from within its own class function.
11. How would you build a class so that it maintains the persistence property of object-oriented programming?
12. Why should the private member functions in a class almost always have void returns and no inputs?
13. Now that we’re working with classes that we write, do I have to include the *iostream* and *fstream* libraries to write output to the screen or to a file? Explain.
14. Suppose we write a class and declare an object of this class in main. What happens if we use the object to call a private member function? Explain.
15. If I declare an array of 2000 *Student* objects (assume *Student* is a class I’ve written), is a class’ constructor function called before or after I start using the *Student* array elements? Explain. Which (if any) class constructor is called?
16. If you are using an array of objects, is it still possible to have overloaded operators and use them with individual array elements? Explain.
17. Why is it good programming style to place the class declaration in a header file (*.h) and the associated code in a source code file (*.cpp).
18. In the previous chapters we wrote many programs using “Ask” functions that ask for program input and returned these values to the calling function. Why did we not call these functions “Get” functions?
19. Explain the difference in the call statement when passing an object’s address (explicitly) versus a reference to function. How are the object’s functions accessed in the called function?
20. Why must the input arguments be unique when you are overloading a class function?

Debugging Problems: Compiler Errors

Identify the compiler errors in Problems 21 to 25 and state what is wrong with the code.

21.

```
#include <iostream>
```

```
using namespace std;
class Ball
{
    private:
        string shade;
    Ball(){ shade = green;}
    WhatColor{}{cout << shade;}
}
int main();
{
    Ball MyBall;
    MyBall.WhatColor();
}
```

22.

```
#include <iostream>
using namespace std;
Class Birds
{
private:
    string name;
    float size;
public:
    Birds();
    void SingASong();
}
void SingASong()
{
    cout << name << "says cheep cheep";
}
int main()
{
    Birds Sparrows[30];
    Sparrows.SingASong[0];
}
```

23.

```
#include <iostream>
using namespace std;
class Printer
{
private:
    int pages = 0;
    int mode = 2;
public:
    Printer();
};
int main()
{
    Printer 4HP2;
```

24.

```
#include <iostream>
using namespace std;
class Photo
{
private:
    string subject;
    double amount; //of light in lumens
public:
    Photo();
    Void SetLight(double x){amount = x;}
}
int main()
{
    Photo OneOfMe;
    OneOfMe.subject = "Meg the cute girl.";
    OneOfMe.SetLight(100.0);
```

25.

```
#include <iostream>
class Dolphin
{
private:
    float size;
    string name;
public:
    Dolphin(){name = "Tosser"; }
    void SetSize(float s) { size = s;}
    bool operator > ();
};
void Dolphin::operator > ()
{
    if(X.size > size) return false;
    else return true;
}
int main()
{
    Dolphin Big, Little;
    Big.SetSize(20);
    Small.SetSize(15);
    if(Big > Little) cout << "bigger";
    else cout << "smaller".
}
```

Debugging Problems: Run-Time Errors

Each of the programs in Problems 26 to 28 compiles but does not do what the specification states. What is the incorrect action and why does it occur?

26. Specification: The following program should set the dimensions of a Cube object to 3 and calculate the volume and surface area.

```
#include <iostream>
#include <cmath>
using namespace std;
class Cube
{
private:
    float side;
public:
    Cube() { side = 0.0; }
    void SetSide(float s) {side = s; }
    float Volume(){ return pow(side,2); }
    float SurArea() { return 8.0 * side*side; }
};
int main()
{
    Cube MyCube;
    float CubeVol, CubeSA;
    MyCube.SetSide(5);
    CubeVol = MyCube.Volume();
    CubeSA = MyCube.SurArea();
    return 0;
}
```

27. Specification: When the object is created, the program below should set the time to HR:MIN:SEC. The `++` operator should increment the minute value. When the fifty-ninth minute is incremented, the HR is incremented and the MIN is reset to 0. *Note:* Only the class functions are presented here.

```
class Time
{
private:
    int hr, min, sec;
public:
    Time(int h, int m, int s) { hr = h; min = m; sec = m; }
    void operator ++ ();
};
void Time::operator ++ ()
{
    min++;
    if(min == 59)
    {
        hr++;
        min = 0;
    }
}
```

28. Specification: The Convert class performs conversions from miles to inches. We need to convert 2.5 miles to inches. (Recall that there are 5,280 feet in a mile, 12 inches in a foot.)

```
#include <iostream>
using namespace std;
class Convert
{
private:
    double miles, inches;
public:
    Convert(){ miles = 0.0; inches = 0.0; }
    Convert(double m){miles = m; }
    double Miles2Inches();
};

double Convert::Miles2Inches()
{
    inches = miles * 5208.0 * 12.0;
    return inches;
}

int main()
{
    Convert MyMiles(2.5), MyOtherMiles;
    double MyInches;
    MyInches = MyOtherMiles.Miles2Inches();
    cout << "\n Total inches are " << MyInches;
    return 0;
}
```

Programming Problems

Write complete C++ programs for the following problems.

29. Write a C++ program that sets up a class Sailboat containing private data for a boat's manufacturer and name, length, beam (width), and draught (water depth required to float the boat). The class has *set* functions for all data as well as a WriteInfo that reports all the Sailboat object data to the screen. It also has a *>* operator that compares Sailboat objects. One boat is greater than another if it is longer. If the boats are the same length, then the larger beamed boat is bigger.

Create a data file of boat data called “boats.txt” so that the first line is the total number of boats in the file; the next three lines are the manufacturer; followed by the name; and then the boat length, beam, and draught. For example, this file contains three boats:

3

McGregor 25
The Sparkly Lady
25 8 3.5

```
Victoria
WayLay
18 6 3
Hobie
Lazy Days
18 8 1.0
```

Your *main* function should set up an array of five boat objects. Place five boat objects in your “boats.txt” file. In *main*, call *ReadFile*, which open your data file and fill your *Sailboat* array with the boat data from the file. It returns a true if it found the file and read successfully, a false if it could not open the file. Report any file problem from *main*. If the array was filled, show the boat information in the order of the data file (unsorted) and then sort the array by using a *BubbleSort* function. Write the boat objects to the screen in the order from smallest to largest boat.

30. In Chapter 5, Problem 32, (page 290) we wrote a program for the C++ International Airport parking garage. We are now going to rewrite the program using a *ParkingGarageAttendant* class. (You may shorten the name of the class to *PGA*.) Please read the Chapter 5 problem. It describes the non-class approach to this situation. For this problem, our *PGA* class has two constructors, the default constructor sets the parking rate to \$2.00 per 30 minutes. The overloaded constructor is passed the dollar and minute values. (For example, maybe parking is more expensive on a holiday, such as \$2.50 per 30 minutes.) The *PGA* class has a *SetTimeIn* function that is passed the integer hour and minutes the parker arrived at the airport. There is also an *AskTimeIn* function, in which the class asks the user for the time he arrived at the airport in HR:MIN format. The *CalculateFee* function has a dual role. Before it calculates the time parked and fee, it accesses the computer’s system time for the hour and minute—which is assumed to be the time out values. *CalculateFee* calls the private *ValidateTimes* function that checks to see that the time in values are reasonable (i.e., hour 0–23, minutes 0–59) and that the time in is before the time out. The *ValidateTime* sets a valid flag accordingly. (The valid flag is a class member, true indicates good data, false, bad.) If the data is good, the time and fee are calculated and a true is returned. If the times are not valid, the *CalculateFee* returns a false. The *CalculateFee* function uses references to return the fee, and total time parked in hours and minutes to the parker.

In *main*, declare one *PGA* object. Begin a “parking” loop that asks the user to enter the arrival time at the airport in HR:MIN format. You may either obtain this time in *main* and set it into the *PGA* object, or have the object ask the parker directly. Remember, write your fee using a “\$” and two digits of accuracy. Your *PGA* class contains *cout* statements only in the *AskTimeIn* function. All other information including invalid times statements are written from *main*.

31. Oh no! Not another Mortgage Calculator problem! Oh yes. Look at what we’ve accomplished with this one problem. We wrote the entire ask, calculate, show results code in *main* in Chapter 2, Problem 46. In Chapter 4, Problem 28 we broke the program tasks into functions. In Chapter 5, Problem 27, we streamlined the data collection by have an *Ask* function that obtained the three inputs. Now we’ll write a C++ Mortgage Calculator class.

The MortCalc class should have one default constructor that initializes the class variables to “safe” values. Of course, you know the variables and equation by heart now. The class should have the principle, interest, years values, as well as total loan cost, total interest paid, and monthly payment. Write the class with individual *Set* functions for the three input values, and individual *Gets* for the three outputs. There should be a *GetFormattedString* that return a string with a complete description of the calculated loan information. This class has a private *Calculate* function that computes all the loan values. There are no *cout* or *cin* statements in this class.

Remember when you write a class, you are not guaranteed that the programmer will call the public functions in any given order. Your programmer could make a *MortCalc* object and then call *GetFormattedString*. Be sure that the constructor does its job and initializes the values to something “safe” so you don’t run the risk of your program crashing or returning “trash” from the *Get* functions. Your *Set* functions will all need to call the *Calculate* function to ensure the data is updated if any one value changes.

Write a *main* function that creates a *MortCalc* object. Present a menu allowing the user to choose to enter data needed for a loan. (The *main* function obtains the values and passes them into the object.) Print the results that *main* gets from the calculator object to the screen. The menu should allow the user to enter all three values, or just one at a time. Each time the user enters a value, the loan information is printed to the screen. There should be a “loan” loop so that our user can calculate all the loan information he desires.

32. We’re going to write a *TakeAQuiz* class. Just as the name sounds, this class allows the user to take a quiz by answering a series of questions. The class selects a question from a group of questions and the user enters his answer. The class tells the user if he is correct (with a “Good!” comment) or not (showing the correct answer). After he is finished answering questions, the class reports how well he did on his quiz. Results include total questions asked, total correct, total missed, and gives a percentage score and a letter grade. Percentage scores need to be calculated with decimal precision, and shown with 2 decimal places of accuracy. The grade scale is: 100–90% A, 89–80% B, 79 – 70% C, 69–60% D, below 60% F. You may round the percentage: 89.9 is rounded to 90%, and an A.

This program uses a *Quiz* class object to perform the tasks of giving, grading, and reporting the quiz information. We are going to base our class so that it uses two vectors, one to store the questions and one to store the answers. Use a string for the topic. An outline of this class is here:

```
class Quiz
{
private:
    //vectors or Q, A and integer randomizer
    //other class data pertinent to the quiz
    void AskAQuestion();
    void ReadDataFile    //asks whether user wants default file or own file
    void GradeQuiz();   //determines the percentage and letter grade
public:
    Quiz();           //reads QuizData.txt for quiz material
```

```

void AskForName();           //uses cout to request user's name
void SetName(string name);  //is passed the user's name
void TakeAQuiz();           //logic for controlling the taking of
                           //the quiz
void WriteQuizStats();      //write quiz statistics to the screen
void GetQuizStats()          //needs to have references for name,
                           //topic, total Qs, raw Q/A,
                           //percent, letter grade
};


```

The format for the quiz material data file contains the topic on the first line, followed by question, and its answer is on the following line. The question should be a complete question that can be shown to the user. Such as:

```

State Capitals
What is New Mexico's state capital?
Santa Fe
What town is California's state capital?
Sacramento
Arizona's state capital is?
Phoenix

```

The Quiz constructor initializes the data. The main function calls the *ReadFile* function, which asks for file selection and reads and fills the Q and A vectors. If there were no problems reading the file, *main* function then call the *TakeAQuiz* function, which has the logic for taking the quiz. It calls the *AskAQuestion* function that contains the logic for asking the “current” question, obtaining the user’s answer, comparing the answer, and reporting right/wrong results (if the user answers incorrectly, show the correct answer.) This function also sets the necessary Quiz class data that keeps track of test results.

When the user has decided to not answer any more questions, the program control returns to *main*. This class is designed so that it can either report quiz information to the screen (using *couts*) or you can obtain it through the *GetQuizStats* function. Therefore, in *main* you need to either obtain the quiz stats from the object and write to the screen or ask the object to write out the quiz results. This data includes the user’s name, quiz topic, raw Q/A data, the grade average, and letter. The GradeQuiz class function determines the percentage and letter score.

For extra credit, write the *TakeAQuiz* function so that it randomly orders the question/answers, and make sure the questions aren’t repeated until all questions have been asked. In order to do this, the function must have a vector of integers that are shuffled and then used to access the question/answer pair. You need to use the *<algorithm> random_shuffle* function to do this. (Research this in the Visual C++ 2005 Express Help. You’ll find examples on how to use this handy tool.) You’ll shuffle the integers and then use the shuffled list (in order) to select the question/answer. Once all the questions have been asked, reshuffle the integer vector, and start again.

33. We’re going to write a program that crashes and your task is to figure out what is causing the crash. We’ll name it Carrie’s Crashing Program (after its original programmer.) Write a simple Date class that contains three integers for

month, day, year, and a string for the name of the month. There should be a vector of strings named MonthNames that will hold the names of the months, “January”, “February”, etc. Write a *GetFormattedString* function that builds a string with month name, day, year, such as “October 27, 2006”. It uses the integer for the month (less 1) to obtain the correct name of the month.

Your Date class should have two constructors. Perform these tasks in the default constructor: set the time to the system time and load the MonthNames vector by calling the *push_back()* function twelve times. Your overloaded constructor is passed the three date values. Assign these values into their class members.

In *main*, create two *Date* objects. Make one Date object using the default constructor. (It should set the date to today’s date.) Make a second Date object using the overloaded constructor and pass in your birthday. Next call the *GetFormattedString* for both objects to obtain the strings of the dates. CRASH! Can you explain where and why you’re getting a crash? Use the debugger and check the data values for both objects. Why does the first *GetFormattedString* function work correctly but crashes the second time you call it?

34. Using the Date class that we built in this chapter, now add these overloaded operators: *>* *>>* *<* *<<* *==* and *+*. The operators will be used to determine “greater” and “less than” and check for sameness. The *>* *<* operators are to evaluate two Dates for their occurrence in the year. One date object is “greater than” another date object if the first date object occurs later in the year. Therefore February 7 is greater than January 10. The year is ignored. (Remember, later is greater.)

The *<<* and *>>* operators includes the year in its evaluation. A date object is greater than (“*>>*”) another date object if the first date object after the second. For example, January 10, 1999 is greater than January 10, 1960. Think of the calendar as a timeline with the “beginning of time” on the far left, and time progressing to the right. Any date to the right of another date would be considered greater.

In this program we’re going to read in a list of dates and populate an array of *Date* objects. We’ll then provide the user a way to analyze the various date information. Build a *DateManager* class. This class has an array of 15 *Date* objects. The constructor initializes class data, and the main function calls a *ReadFile* member function which asks the user for the name of the date file. The file contains 15 dates and their descriptions. Here’s an example of a 5-item date file:

Graduation from CNM Community College	(description is first)
4/25/2008	(followed by the date)
Mom and Dad's Wedding Date	
2/10/1966	
Bob's Birthday	
2/2/1968	
First day of school	
8/29/2006	
The day that TigerLily (new puppy) came home	
5/13/2005	

In this program, the DateManager class has a public *Menu* function that shows your user a list of choices. The menu lets the user show the current list of dates in the array, search the list for a specific date or event, sorts and shows the dates in chronological order based on year order or chronological order based on occurrence during the year. Then menu should also have a “Return to main” option, which then terminates the program. (DateManager has class member functions that perform each task. Results are reported by the Date Manager class. (It will call the array elements’ *GetFormattedString* function.) For example, if you wanted to see the events in chronological order, the results would be:

```
Mom and Dad's Wedding Date: February 10, 1966  
Bob's Birthday: February 2, 1968  
The day that TigerLily (new puppy) came home: May 13, 2005  
First day of school: August 29, 2006  
Graduation from CNM Community College: April 25, 2008
```

But if you wanted to see the results based on occurrence during the year they’d be:

```
Bob's Birthday: February 2, 1968  
Mom and Dad's Wedding Date: February 10, 1966  
Graduation from CNM Community College: April 25, 2008  
The day that TigerLily (new puppy) came home: May 13, 2005  
First day of school: August 29, 2006
```

In order to search for a specific date or description, you will need to create a temporary *Date* object, obtain the data from the user and set it into that *Date* object. For the date, ask the user to enter a date in M/D format (i.e., look for all occurrences on that month and day). Your program should search on either full or partial description (i.e., “Mom and Dad” or “TigerLily”). Use the “==” to compare dates (month, day) and the “+” to compare descriptions. In both cases, you’ll be using a temporary *Date* object, and your results will show all occurrences.

Two notes: for the two types of sorting required for this program, you’ll need to create two private sort functions based on the *Bubble Sort* function. Remember, there are NO *cout* statements in your Date class, but many *cout* and *cin/getline* statements in your DateManager class. Construct the DateManager in main, read the file and if no problems occur, then call the *Menu* function.

35. Sophisticated vending machines can handle dollar inputs and give the correct amount of change. To see one of these machines in action, visit a US Post Office and buy stamps from its vending machine. You can insert a \$10 bill to purchase a single first class stamp. Your change will be in coins, including the \$1 coin.

We’re going to write a program that tests a CoinChanger class. The CoinChanger counts out change using dollars, quarters, dimes, nickels, and pennies. Write a class named CoinChanger. The CoinChanger has two constructors, the default sets the number of each coin to 20, except for pennies, which has 100. (20 dollars, 20 quarters, 20 dimes, 20 nickels, and 100 pennies.) The overloaded

constructor is passed the number of coins. The Changer needs to keep a running total of its money, as well as keeping the count of all coins.

When a *CoinChanger* object is made, it opens an output file (ask the user for the name) and writes your name, a program title and gives total number of coins and total amount of money. The CoinChanger should have a private *WriteLog* member function that logs the information to the output file.

The crux of the CoinChanger's work is performed in the *MakeChange* function that is passed the total dollar amount that is to be returned to a customer. (It is passed as a single float value. For example \$2.56 would be passed as 2.56.) The *MakeChange* function returns a boolean indicating whether or it is able to make the requested change. If it can, it then reports (using *cout* statements) the total number of each coin returned to meet the amount of change requested. You must use the least number of coins to make change. Log the time, the requested change, and coins used, and then the coins remaining. For ease of programming we'll assume that the Changer is unable to make any change if it runs out of one coin. Check to be sure the total amount of money in the Changer is sufficient to cover the requested change. After all, you can't give more money than you have!

For example, if the default CoinChanger is asked to make \$0.88 in change, it gives 3 quarters, 1 dime, and 3 pennies. (Remaining coins are 20 '\$'s, 17 Q's, 19 D's, 20 N's, and 97 P's.) If the changer was then asked to return \$4.34 it gives 4 dollar coins, 1 quarter, 1 nickel, and 4 pennies. (Remaining coins are 16 '\$'s, 16 Q's, 19 D's, 19 N's, and 93 P's.)

Hint: sometimes beginning programmers have found themselves 1 cent off when testing their program. Remember that floats and double variables may store value in a different form than you expect. You may think one penny is 0.0100000 (float) when it is actually stored as 0.0099999. You may need to add a little fudge factor to your money value to bump it up a bit. You can do this by adding 0.000001 to a floating point value.

36. In Chapter 6, Problem 35 (page 362) you wrote a program that asked the user to enter a month and year, and it produced an output file showing a calendar for that month. (Depending on your programming expertise, you may have asked the user to tell the program what day of the week was the first of the month, or you found an algorithm to do that task for you.) Now is the perfect time to rewrite this program so that the monthly calendar file is built with a C++ class.

Create a *CalendarMonth* class that contains the six by seven integer grid, along with string arrays (or vectors) containing the twelve months and seven days of the week. Your class should contain *set* functions for the user's desired month and year (and day of week if needed). The class may also contain an *Ask* function that uses *cout* statements to ask the user for the month and year (and day of week) values. You may be thinking that this is redundant coding, but remember your class is offering the programmer two ways to obtain the month and year data. The one class constructor should set the default month to January, and year to the current year. There is a private *WriteMonth* function that does the work of writing the Month_Year.txt file containing the calendar.

Let's make this interesting and add a few more features to this class. Let's add a month fact for each month. This fact is written below the monthly calendar. Each fact should reflect some seasonal or activity related item to the month. Also, for each month, identify one day that should be highlighted using * * around the date. Indicate below the calendar that special day. For example, February 2007 might look like this:

February 2007						
Sun	Mon	Tue	Wed	Thu	Fri	Sat
				1	2	3
4	5	6	7	8	9	10
11	12	13	*14*	15	16	17
18	19	20	21	22	23	24
25	26	27	28			
* Valentine's Day *						

Did you know that February is the shortest month of the year?

- 37.** Problem 34 in Chapter 6 (page 361) utilized the Bingo functions we wrote in Programs 6-11 and 6-12 and created a technique for marking the called Bingo numbers. We also had a *CheckForXBingo* function that returned a true or false status indicating if a Bingo had been reached. Now let's combine these functions and two-dimensional array data into a BingoCard class. We'll build our BingoCard class functionality to match how a real Bingo game is played!

Create a BingoCard class that contains the necessary "smarts" so that it knows how to mark and check itself, given the Bingo game "called" numbers. Our class has a default constructor that sets the winning Bingo pattern to any row, any column or any diagonal. There is also a public *SetBingoPattern* function that is passed a string. If a pattern is set, the default patterns are no longer winning patterns. Our class has the ability to check for the X pattern and square pattern (top and bottom rows, and left and right columns). Use the strings "X Bingo" and "Square Bingo" to identify these in the class. There are three private functions for checking the Bingo card for these patterns. The private data includes the five-by-five integer grid, along with the associated grid for holding the marked spots on the card. The class should have a *SetCalledNumber* function that is passed the Bingo numbers that are called during a game, such as B17, N42, etc. (We'll just use the integer value for our program, don't worry about the BINGO letters.) This *Set* function returns a true/false for Bingo status, as well as writes a message if it has a Bingo. The class has the *FillCard* and *DisplayCard* members along with a constructor function that initializes the card and marked grid data. When a Bingo number is passed into the class, the card is "marked," displayed and checked for a Bingo.

In *main* make two Bingo card objects and begin a loop that "calls" the Bingo numbers. Pass that number to both cards and report if a Bingo has been won. Here is a partial section of the *main* function. (Can you figure out why we need two individual *if* statements instead of an *if-else if*?)

```
BingoCard blue, red;
//tell the cards what pattern we're playing
```

```

blue.SetBingoPattern("Square Bingo");
red.SetBingoPattern("Square Bingo");

//Here is the play loop. Very simple.
do
{
    int playNumber = GetNumber(); //generates non-repeating
                                  Bingo #'s
    if( blue.SetCalledNumber(playNumber) )
    {
        cout << "\n The blue card has a Bingo!
Game over! \n";
        gameOver = true;
    }
    if( red.SetCalledNumber(playNumber) )
    {
        cout << "\n The red card has a Bingo!
Game over! \n";
        gameOver = true;
    }
    cin.ignore(); //pause for an enter key
}while( gameOver == false);

```

The BingoCard's *SetCalledNumber* function does the lion's share of the work! Here is an outline of this function. Read it carefully as it provides more insight to the necessary and useful class member.

```

bool BingoCard::SetCalledNumber(int play)
{
    playValue = play; //assign into class member
    //first we mark the card
    MarkCard();
    //next we check for the Bingo
    //use the utility function CheckBingo,
    //it knows which check function to call
    //and sets the bBingo flag to indicate if we have a winner
    CheckBingo();
    //show the card
    DisplayCard();
    //write winning statement if appropriate
    if(bBingo)
        cout << "\n BINGO!!! BINGO!!! \n";
    return bBingo;
}

```

Your *main* function can be structured to have a play again loop, and perhaps even to ask the user to enter the Bingo patterns she wishes to play. Remember, when you build classes in any object-oriented language, you want to model the real-world scenario as closely as you can. Playing Bingo gives us an excellent example for this task!

38. The C++ Express Train has a wonderful website that allows its customers to book seats on-line for exceptionally low fares. The train runs between Chicago and Albuquerque. For each train, there are a certain number of BargainSeats, RegularSeats, and LastMinute seats. One-way ticket prices are as follows. BargainSeats, \$39; RegularSeats, \$79; LastMinute, \$129. The fare is based on the booking date of the reservation and the date of the trip (or departure date if it is a round trip). Fares are calculated this way: if booked 30 days in advance, the fare is a bargain, if booked 2 weeks in advance (i.e., 14 days), it is regular fare, anything closer is last minute.

In this program, we're going to write a class that represents the reservation data for an individual or group traveling between Chicago and Albuquerque. Here is the CPPTrainTickets class declaration. The Date class here is the Date class we've built in this chapter. (You may add more private members to the tickets class if you wish.)

```
class CPPTrainTickets
{
private:
    string reservation;      //passenger name for this reservation
    int numberofPass;        //number of passengers booked under this
    Date book, dep, ret;     //pertinent date
    bool bRoundTrip;
    string DepartureCity;
    float fare;
    void CalculateFare();

public:
    CPPTrainTickets();
    void SetNumberofPass(int n);           //how many people?
    void SetDepartingCity(string city);    //either ABQ or CHI
    //overload SetDates for either one way or round trip
    //pass in Date objects
    void SetDates(Date bk, Date dpt, Date rtn);
    void SetDates(Date bk, Date dpt);
    void SetReservationName(string name);
    float GetTotalFare();
    string GetReservationDescription();
};
```

Your job is to write the CPPTrainTickets class. Once it is written, test this class by writing a *main* function that declares one CPPTrainTickets object. You should then begin a “reservation loop” that ask the user for pertinent data for making a train reservation. Then call GetReservationDescription which returns a string that contains the reservation name, number of passengers, travel date(s), destination, and total fare. *Note:* Code the CPPTrainTickets class so that if the user has submitted an invalid date, the reservation description string states that it was unable to determine the fare due to an invalid date. (Show the bad date.)

39. When you design an irrigation system for your yard, the components usually consist of emitters (such as sprinker heads and drippers), pipes, connectors, and valves. In this program we're going to write an `IrrigationEstimator` class, which contains a vector of `IrrigationComponents`. This program reads a data file of required components for a landscaping job, and then writes an invoice (output file) summarizing the irrigation parts, cost, and gallons per hours flow rate.

Let's start at the component level. You'll have a data file (let the user enter the name, the program will ask for it) that contains the various components. It is organized in this manner. Here is a sample.

```
Johnston Residence      (first line is the name of job)
4 gph Dripper          (name of component)
Emitter (example types may be Emitter, Pipe, Connector, Valve)
4                      (gallons per hour flow, if appropriate)
0.15                  (unit cost $)
10                     (number required for job)
Adjustable Circular Head
Emitter
30
5.25
8
8' PVC Pipe, White
Pipe
0                      (0 if gallons per hour is non-applicable)
4.95
18
Long Range Pulse Sprinklers
Emitter
40
10.95
3
L-Connectors, White
Connector
0
0.18
25
```

You may assume that there aren't duplicate component names in the file, but expect that there are duplicate component types (i.e., Emitters, Connectors, etc.). Also, don't assume that this program will only work with the component types shown here. There may be other types in the input file.

There is a class called `IrrigationComponent` that represents each group of irrigation components found in the file. The class contains strings for the type and name, an `int` for required number, floats for the gallon per hour and cost values. There is a default constructor, and `Set` functions for the data. There are `Get` functions for unit cost, total cost, total GPH, and name. Note: There are no `cout` or `cin` statements in this class!

The IrrigationEstimator class does the majority of the work in this program. It contains a vector of *IrrigationComponent* objects. Because we don't know how many sets of components are in the file, a vector is what we need to use. (Hint: the vector declaration inside the *IrrigationEstimator* is vector, such as `<IrrigationComponents> vComp.`) Among other things, the class also includes a string for the name of the job, and a business name for the estimator. The *Read()* function asks the user to the name of the component data file, it reads and fills *IrrigationComponent* object, which is pushed into the vector. The Estimator contains private *Calc* functions for total cost, total gallons per hour. The *WriteInvoice* function asks the user for the invoice file-name and then writes the invoice file.

The invoice file must contain the job description, and an itemized list of components including unit cost and total cost for each component. There is a total cost at the bottom, as well as a total gallons per hour, and total parts for this system. You'll need at least two more vectors to help create this invoice.

Here is a sample of an invoice file based on the data file from above:

Job: Johnston Residence					
Item Type	Name	Unit Cost	Number	Total Cost	
Emitter	4gph Dripper	\$ 0.15	10	\$ 1.50	
Emitter	Adjustable Circular Head	\$ 5.25	8	\$ 42.00	
Pipe	8' PVC Pipe, White	\$ 4.95	18	\$ 89.10	
Emitter	Long Range Pulse Sprinkler	\$10.95	3	\$ 32.85	
Connector	L-Connectors, White	\$ 0.18	25	\$ 4.50	
Totals				64	\$169.95
Total Emitter	21				
Total Pipe	8				
Total Connector	25				
Total Gallons Per Hour flow rate for this system: 400 GPH					

40. Our groundskeeper at the C++ Golf and Spa Resort has to maintain the lake levels for all the lakes at the golf course. It has eight lakes on the course. The water loss for each lake varies because of the location, sun exposure, evaporation due to fountains (and C++ Fish Showers), and the amount of water lost from golf balls that strike the water (thus splashing water out of the lake). Each lake is equipped with a C++ Pump. Review Problem 37 in Chapter 4 (page 242). We're going to write a program that will give him everything he needs for maintaining the lake levels. Our new program reads a file of lake data and then writes a file that contains the length of time needed to run each pump.

To get started, write a Lake class that has four private data items, a string for the name/location, a float for the lake's acreage, a float for the drop in inches and a float for the calculated gallons required to refill the lake. There are *Set* functions for the lake's name, acreage and drop data member. There is a private *CalculateGallons* function, which determines the refill volume given the inches and acreage. The Lake class has public *GetLakeDescription* that returns a string with the name and acreage information. There also is a *GetPumpTime* that returns a string containing the lake data and required gallons and pumping time (in minutes). When the *Lake* object receives the drop data, it automatically calculates the required gallons and time.

Now we come to the fun part of the program! Write a *LakeMgr* class that manages the lake information for our groundskeeper. The *LakeMgr* has an array of eight Lakes. The *LakeMgr* contains a *ReadFile* function that reads the data file containing lake information (see below) and fills the lake objects that are stored in the *Lake* array. The *WritePumpData* function loops through the eight lakes, obtaining the lake and pump time information and writes this to an output data file. The manager asks the user for both the input and output file names. Your *main* function should create a *LakeMgr* object, call the *ReadFile*, and then *WritePumpData*. Easy breezy, don't you think?

You'll have to create your own data file for this program. The format of the input data should have the lake name/location on one line, followed by the size, and then drop in inches for the week. Here is a sample of Lakes.txt, containing the first three lakes:

```
Cottonwood Lake on 3rd Hole      (first line is the name/location of  
                                         the lake)  
0.37                               (second line is the lake's acreage)  
1.5                                (third line is the lake drop in inches)  
Heron Walk Lake on 7th Hole  
1.25  
1.6  
Golfball Magnet Lake on 9th Hole  
2.5  
1.4
```



8

Inheritance and Virtual Functions

KEY TERMS AND CONCEPTS

base class
base class access specifiers
child class
commercial off-the-shelf (COTS) software
compile-time polymorphism
constructor rules for inheritance
derived class
destructor rules for inheritance
inheritance
is a relationship
multiple inheritance
parent class
passing parameters to base constructors
private specifier
protected specifier
public specifier
pure virtual function
run-time polymorphism
virtual functions

CHAPTER OBJECTIVES

- Explain the concept of inheritance in C++ programs.
- Present the basic format required for deriving new classes from base classes.
- Explain how the different access specifiers dictate properties of members in the derived classes.
- Discuss briefly multiple inheritance schemes.
- Explain what constructor and destructor functions are called in C++ inheritance.
- Introduce the concept of polymorphism and virtual functions.
- Demonstrate how virtual functions are used in base and derived classes.



Parents and Children

The Oxford Dictionary of Current English defines *inherit* as "(1) receive property, rank, title, etc., by legal succession. (2) derive (a characteristic) from one's ancestors."¹ Webster's Ninth New Collegiate Dictionary defines inheritance as "1 a: the act of inheriting property; b: the reception of genetic qualities by transmission from parent to offspring; c: the acquisition of a possession, condition or trait from past generations."²

What do these definitions have to do with C++? Imagine having the ability to create a class (a **parent class**), then derive a new class (a **child class**) from the parent. This child class has the properties and characteristics of the parent. The programmer can add additional components unique to the child class. It is time to wade out into deeper programming waters to see if all our object-oriented and C++ programming skills will keep us afloat.

child class

same thing as a derived class, a new class that inherits properties and functions from another class

parent class

a class that is used as a base class, a class whose properties and functions are passed down to a new customized class

8.1

Why Is Inheritance So Important?

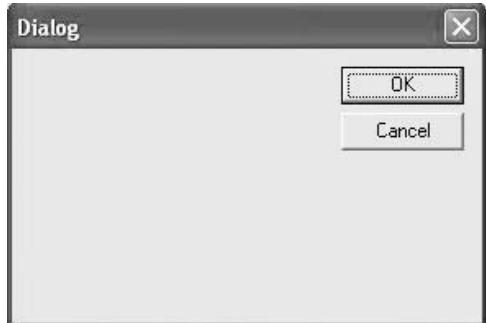
Ice Cream Dialog Example

Most beginning C++ programmers are very computer savvy, having spent time on computers for many years. They wend their way through a program with little trouble, including menu navigation, mouse actions and interacting with programs through dialog boxes. Have you ever noticed that most dialog boxes are essentially the same? They have a frame with user interface controls, a title bar as well as OK and Cancel buttons.

One reason C++ programmers like the language is because of the many pre-built classes (vector, string, queue, etc.) Many C++ classes have been built, and are intended to be starting points for developers, so that the developer can customize the class as needed. The CDialog class is one such class. It is provided to the Windows programmer in Microsoft Visual C++ 2005. (The JDialog class is provided for the Java developers, and wxDialog is found in wxWidgets.)

¹The Oxford Dictionary of Current English, Oxford Press, 1992.

²Webster's Ninth New Collegiate Dictionary, 1984.

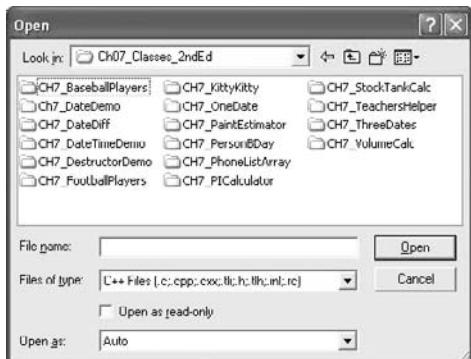
**Figure 8-1**

The initial dialog box without any user interface components.

The starting point for dialog boxes is the same. Figure 8-1 shows a plain dialog box that is created by Visual C++. You can see that it has a title bar, an “X” to close the frame, and OK and Cancel buttons.

Microsoft Visual C++ also provides the OpenFileDialog class, Figure 8-2, which is a customized version of the CDialog class. Someone in Microsoft started with the CDialog and then customized it so the programmer could put it into a program to help the user select a file to open. The OpenFileDialog gives the C++ programmer a way to set the file type (i.e., file extensions, such as .c .cpp), and designates the path (Look in:) and Open as: command.

What if we wanted to write a program and build our own dialog box? Because there is a CDialog class available, we do not have to start from scratch—we start at the CDialog class, create a new class using it as a base class, and customize our new class as needed! We don’t have to worry about making a frame and title bar, all we have to do is lay in the controls, and hook in the code so that our program listens for the controls to be pressed. Another way to think of these base class/derived class relationships is this: when you “inherit” from a base (parent) class, your derived (child) class receives the data and functions of the parent class. So someone built the CDialog class that provides the window, title bar, buttons. When you use it to create a new class, you get these things for

**Figure 8-2**

The OpenFileDialog is a customized CDialog class.

**Figure 8-3**

The IceCreamDialog class is a custom-built dialog box. The starting point is CDialog.

free. Figure 8-3 shows the IceCreamDialog, which is a class that is derived from CDialog class.

In C++ (or any object-oriented language) we are able to create a new class from an existing class. We may use a class, such as CDialog, and inherit the “dialog” parts and pieces. Our new class obtains the frame, title bar, the “X”, the buttons. All we have to worry about is adding our components so that it functions in the manner we desire. Our IceCreamDialog is a customized sub-class of CDialog. We used inheritance to build this class. Isn’t that slick?

We’re going to use simple examples in this chapter to illustrate how inheritance works and cover details such as constructors, destructors, as well as overloading and overriding functions. We introduce virtual functions and polymorphism, too. These concepts are used extensively in the advanced C++ programming courses and in graphical user interface development. They are presented here for completeness. Also, when you, the beginning C++ student sees how it all works together perhaps you’ll be motivated to continue your study of C++!

Counter Class Example

To get us started learning about inheritance, we have designed a very simple Counter class. This Counter class has an integer count variable. The constructor zeros the count value. An overloaded operator, “`++`”, increments the count value, and a `GetCount` function returns the count value. We have also a `PrintCount` and a `SetCount` function. Notice that all of the class functions are one-line long. The entire class can be described in the declaration, located in the file Counter.h, is shown below.

Program 8-1

```

1 //File: Counter.h
2 //A simple class that has an integer counter.
3
4 #ifndef _COUNTER_H
5 #define _COUNTER_H
6
7 #include <iostream>

```

```
8  using namespace std;
9
10 class Counter
11 {
12 private:
13     int count;
14 public:
15     Counter(){count = 0;}
16     void operator ++(){ ++count; }
17     int GetCount() { return count; }
18     void SetCount( int c ) { count = c; }
19     void PrintCount() { cout << "\n The count is " << count; }
20 };
21
22 #endif
```

We have a simple *main* function that tests our Counter class. We make an object named *HowMany*. With this object we use the *++* increment operator, *Set* and *Print* functions. Look at the output. You will see that our Counter class works correctly.

Program 8-1

```
1 //File: CountDriver.cpp
2 //Sample program with a counter object
3
4
5 #include "Counter.h"
6 #include <iostream>
7 using namespace std;
8
9 int main()
10 {
11     //make a Counter object
12     //its constructor sets HowMany count = 0
13
14     Counter HowMany;
15
16     cout << "\n Sample program with class Counter \n";
17
18     HowMany.PrintCount();
19
20     cout << "\n Increment HowMany twice: ";
21     ++HowMany;
22     ++HowMany;
23
24     HowMany.PrintCount();
25 }
```

```

26     cout << "\n Now set the count back to zero. ";
27     HowMany.SetCount(0);
28
29     HowMany.PrintCount();
30     cout << "\n\n All finished counting! \n";
31
32     return 0;
33 }
```

Output

Sample program with class Counter

```

The count is 0
Increment HowMany twice:
The count is 2
Now set the count back to zero.
The count is 0
```

All finished counting!

One day you discover that you need a customized version of your Counter class. You need to keep track of the name of the person and description of what that person is counting, as well as decrement the count. (Remember, the Count contains the `++` operator.) Of course, you might be tempted to just grab the Counter.h file and modify it. In the spirit of object-oriented programming, however, we'll use the Counter class as a starting point to derive a new, customized Counter class named DeluxeCounter. (It is the same idea as we did used to derive our Ice-CreamDialog class from the CDialog class.)

8.2

Inheritance Basics

The class relationship that models **inheritance** is the **is a relationship**. The **base class**, typically, is a general-purpose class. The **derived class** is a special case of the base class. The phrase “*is a*” describes how the classes are related. The DeluxeCounter “*is a*” Counter. The relationship is not true in the reverse. The Counter is not a DeluxeCounter.

When a new class is derived from a base class, the new class “inherits” (i.e., automatically receives) protected and public members of the base class. The DeluxeCounter is derived from the Counter class. It inherits the protected and public members of Counter—there is no need to write these members in the new class declaration. The format for class inheritance is shown below:

```

class BaseClass
{
    // members of the base class
};
```

inheritance

a new class is created from an existing class—this new class contains data and functions from the existing class

is a relationship

a class relationship in which one class is derived from a second class (the second class is a special case of the first class)

base class

a general-purpose “parent” class and new classes are derived from the base class

derived class

new class or “child class” created by inheriting data and functions from a base or parent class

```
class DerivedClass : access_specifier BaseClass
{
    // members of the derived class
    // inherit protected and public members of the base
};
```

The line:

```
class DerivedClass : access_specifier BaseClass
```

contains a colon (:) which is C++'s way of saying this BaseClass is the parent for the new DerivedClass. The access specifier, usually the public specifier, dictates access properties for the inherited members. There will be more on this later in the chapter.

Once you have built the new, derived class, you can create derived class objects and use them in the normal fashion.

Counter and DeluxeCounter Example

Our Counter class allows the programmer to increment, set, and get the count value and print the value to the screen. The decrement function along with new class variables will be added in the derived class, DeluxeCounter. Recall the original Counter class, which is shown below.

```
class Counter
{
private:
    int count;                                //this data member is private!
public:
    Counter(){count = 0;}
    void operator ++(){ ++count; }   // add one to the count
    int GetCount() { return count; }
    void SetCount( int c ) { count = c; }
    void PrintCount () { cout << "\n The count is" << count; }
};
```

The format for declaring a derived class for DeluxeCounter is shown here:

```
class DeluxeCounter : public Counter
{
private:
    //add additional data members here that DeluxeCounter needs
public:
    //add public functions to support new data and functionality
};
```

The first line in the declaration, class DeluxeCounter : public Counter, states that the class DeluxeCounter is derived from the class Counter. The term *public* is a *base class access specifier*. (We examine its purpose in Section 8.3.) The public and

protected members of Counter are inherited by DeluxeCounter, but the base constructor functions are not. The DeluxeCounter class has two new data members, a new overloaded operator function that decrements the count value, and set functions. Here is the DeluxeCounter.h file, which shows our new, derived class.

Program 8-2

```
1 //File: DeluxeCounter.h
2
3 #ifndef _DELUXE_COUNTER_H
4 #define _DELUXE_COUNTER_H
5
6 #include <iostream>
7 #include <string>
8 using namespace std;
9
10 #include "Counter.h"
11
12 //This class inherits all the protected
13 //and public members of the Counter class
14
15 class DeluxeCounter : public Counter
16 {
17 private:
18     string name, what;
19
20 public:
21     DeluxeCounter();
22     DeluxeCounter(string n, string w, int c);
23
24     //set name and what
25     void SetName(string n){ name = n; }
26     void SetWhat(string w){ what = w; }
27
28     //decrement counter too
29     void operator --(){ --count; }
30
31     //new version of print, includes name/what
32     void PrintCount();
33 };
34
35 #endif
```



We cannot forget one detail in all this inheritance business: the fact that in our original Counter class the variable “count” is private. When a class has private members, only that class’ functions may access or change this variable. Private data members are not “passed on” to derived (child) classes. We need our DeluxeCounter class to inherit the data members of Counter so that all the DeluxeCounter members may



access “count.” Therefore, we must change the access specifier in Counter from private to protected. This done, our DeluxeCounter inherits the data variable count. Here is the one change we make to our Counter class, located in Counter.h:

```
class Counter
{
protected:      //  <-- protected, not private
    int count;
public:
    Counter() { count = 0; }
//etc, no other changes in the code
};
```

Protected Members

The ability of a new class to inherit functionality from a base class is a powerful feature in C++ programming; however, if the data of the base class is all privately declared (as in the original Counter class above), the derived class will have extra work for obvious tasks. We do not want to make the data public because that action defeats the data-hiding principle of object-oriented programming. There is a third specifier in C++ designed just for inheritance.

The protected access specifier provides the programmer greater flexibility when he is working with inherited classes. When a class member is specified as **protected**, that member is inherited by the derived classes, but the member is not accessible outside the base, or derived class. The Counter class can declare the count variable as protected, the DeluxeCounter class inherits it, but the world cannot access it. Table 8-1 summarizes the three access specifiers in C++.

Now let’s look at our DeluxeCounter class in more detail. The majority of the functions are one-liners and can be seen in DeluxeCounter.h. The *DeluxeCounter* functions located in the associated *.cpp file are short and sweet. Notice how we write these functions using the class variables count, name, and what. The DeluxeCounter inherited count from Counter, so it’s part of that class.

protected

an access specifier,
protected members
are inherited by de-
rived class

■ TABLE 8-1

Private, Protected, and Public Access

Keyword	Access Specifier Definition
private	The private members in a class can be seen, used, assigned, and/or changed only by other members of the class. Private class members are not accessible by the object nor are they inherited by the derived class.
protected	The protected members can be seen, used, assigned, and/or changed by other members of the class. Protected class members are inherited by the derived class. Protected members are not accessible by the object.
public	The public members may be seen, used, assigned, and/or changed by all class members, as well as accessible by objects. Any part of the program that has access to the class object can access public members via the object.

Program 8-2

```
1 //File: DeluxeCounter.cpp
2
3 #include "DeluxeCounter.h"
4
5 DeluxeCounter::DeluxeCounter()
6 {
7     //automatically calls Count constructor
8     //first, so count is getting set to 0 there
9     name = "";
10    what = "";
11 }
12 DeluxeCounter::DeluxeCounter(string n, string w, int c)
13 {
14     //we assign values here
15     count = c;
16     name = n;
17     what = w;
18 }
19
20 //new version of print, includes name/what
21 void DeluxeCounter::PrintCount()
22 {
23     cout << "\n " << name << " has " << count
24     << " " << what << ".";
25 }
```

In the *main* function we create two *DeluxeCounter* objects, named *superDooper* and *topNotch*. The *superDooper* object will be used to count Donna's chickens, and the *topNotch* object will count Don's tractors. Before we look at the code any further, Figure 8-4 shows the pick list that is presented to the programmer when he uses the *superDooper* object and dot operator. The *superDooper* object is a *DeluxeCounter*, and was derived from *Counter*. Notice that the list shows the combination of *Counter* and *DeluxeCounter* members! The operator *++* (from *Counter*) and operator *--* (from *DeluxeCounter*) are side-by-side in the list. The private variables “name” and “what” have a lock beside them, the “count” shows the key, indicating protected status, and the other functions are public.

The *main* function and program output is seen here. We practice using the two class constructors in *main* and use the two operators. We only need to include the *DeluxeCounter.h* file, because it fully defines the *DeluxeCounter* object. (It in turn includes the *Counter.h*.) This is a silly program, but it certainly shows inheritance in action! Examine the code and output below.

Program 8-2

```
1 //File: DeluxeDriver.cpp
2
3 #include "DeluxeCounter.h"
```

```
4
5  #include <iostream>
6  #include <string>
7  using namespace std;
8
9  int main()
10 {
11
12     cout << "\n The Deluxe Counter Program.\n";
13     //create two of these high-class counters
14
15     DeluxeCounter superDooper;
16
17     superDooper.SetCount(75);
18     superDooper.SetName("Donna");
19     superDooper.SetWhat("chickens");
20
21     DeluxeCounter topNotch("Don", "Tractors", 15);
22
23     //see what we've got
24     superDooper.PrintCount();
25     topNotch.PrintCount();
26
27     cout << "\n\n Time to fix dinner. ";
28     -- superDooper;
29
30     cout << "\n Time to go to the farm auction.\n ";
```

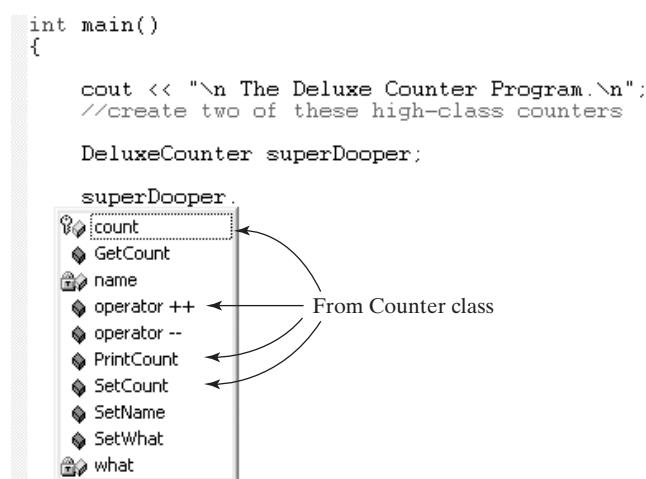


Figure 8-4

The pick list for a *DeluxeCounter* object.

```
31     ++ topNotch;
32     ++ topNotch;
33
34     superDooper.PrintCount();
35     topNotch.PrintCount();
36
37     cout << "\n Don't run over your chickens with"
38         << " your new tractors! :-) \n";
39
40     return 0;
41 }
```

Output

The Deluxe Counter Program.

Donna has 75 chickens.

Don has 15 Tractors.

Time to fix dinner.

Time to go to the farm auction.

Donna has 74 chickens.

Don has 17 Tractors.

Don't run over your chickens with your new tractors! :-)

Employees, Bosses, and CEOs

Let's examine a second example before delving into the object and inheritance details. We begin by building a class for employee information (name, social security number, department, and salary). From there, we derive a new class for boss information. The boss is an employee, who receives a yearly bonus. Lastly, we derive a CEO (chief executive officer) class. The CEO is a boss; the boss is an employee. The CEO class has data to keep track of the stock options that the head of a company receives as additional compensation. The employee is the boss's base class and the boss is the CEO's base class.

Starting with basic employee information for the class Employee, we specify the data that is normally private is protected. This specification allows the derived class to inherit all of the class data. We have one constructor and an *Ask* and *Write* employee information function. The constructor function initializes the string values to "" and salary to zero. The *Ask* and *Write* functions ask for information from the user and write it to the screen, respectively.

Program 8-3

```
1 //File: Employee.h
2
3 #ifndef _EMPLOYEE_H
4 #define _EMPLOYEE_H
5
```

```
6  #include <string>
7  using namespace std;
8
9  class Employee
10 {
11 protected:
12     string name,SSN, dept;
13     float salary;
14 public:
15     Employee();
16     void AskEmpInfo();
17     void WriteEmpInfo();
18 };
19
20 #endif
```

The associated Employee.cpp file shows the class functions.

Program 8-3

```
1 //File: Employee.cpp
2
3 #include "Employee.h"
4 #include <iostream>
5 #include <string>
6 using namespace std;
7
8 Employee::Employee()
9 {
10     name = "";
11     SSN = "";
12     salary = 0;
13     dept = "";
14 }
15
16 void Employee::AskEmpInfo()
17 {
18     cout << "\n Enter the employee's name    ";
19     getline(cin,name);
20
21     cout << "\n Enter the employee's SSN    ";
22     getline(cin,SSN);
23
24     cout << "\n Enter the dept code ";
25     getline(cin,dept);
26
27     cout << "\n Enter the yearly salary   ";
```

```
28     cin >> salary;
29     cin.ignore();
30 }
31
32 void Employee::WriteEmpInfo()
33 {
34     cout << "\n Employee: " << name << "\n      SSN: " << SSN;
35     cout << "\n      Dept: " << dept << "\n      Salary: $" << salary;
36 }
```

The boss is an employee. Therefore, we derive a new child class, *Boss*, from the *Employee* class. Our *Boss* class contains a new piece of data that the *Employee* class doesn't have—the yearly bonus. Our *Boss* class inherits all the *Employee* data, i.e., name, SSN, department, and salary. The *Boss* class inherits also the *Employee*'s *Ask* and *Write* functions but these functions only work with employee data. Therefore, we'll place *Ask* and *Write* functions in the *Boss* class too. The data in *Employee* is protected, so the *Boss* class inherits the protected and public members.

Program 8-3

```
1 //File: Boss.h
2
3 #ifndef _BOSS_H
4 #define _BOSS_H
5
6 #include "Employee.h"
7
8 class Boss: public Employee
9 {
10 protected:
11     float bonus;
12 public:
13     void AskEmpInfo();
14     void WriteEmpInfo();
15 };
16
17 #endif
```

The new and exciting portion of the *Boss* code is found in the *Ask* and *Write* functions. For a *Boss* object, we still need to ask for the name, SSN, department, and salary, but *Boss* asks also for the bonus. There is no need to duplicate the *Ask* and *Write* code from the *Employee* class. All we need to do is just call the *Employee*'s *Ask* and *Write* function. Study this small *Boss.cpp* file. Note how the first line in both the *Ask* and *Write* functions make a call to the *Employee*'s (i.e., the parent's) *Ask* and *Write* function. No need to duplicate the work when the parent class is taking care of part of the class' work.



Program 8-3

```
1 //File: Boss.cpp
2
3 #include <iostream>
4 using namespace std;
5
6 #include "Boss.h"
7
8 void Boss::AskEmpInfo()
9 {
10     Employee::AskEmpInfo();      //ask for name, SSN, dept, salary
11     cout << "\n What is my bonus this year?$$$$ ";
12     cin >> bonus;
13     cin.ignore();
14 }
15
16 void Boss::WriteEmpInfo()
17 {
18     Employee::WriteEmpInfo();   //writes name, SSN, dept, salary
19     cout << "\n     Bonus: $" << bonus;
20 }
```

Taking inheritance one step further, we now create a CEO class for the company's chief executive officer. Our CEO class is a child class of the Boss class (which is a child class of the Employee class). Our CEO needs to have all the data in the employee and boss classes, as well as the CEO specific stock option data. We place an *Ask* and *Write* function in the CEO class, because it needs to Ask for and Write CEO specific data.

Program 8-3

```
1 //File: CEO.h
2
3 #ifndef _CEO_H
4 #define _CEO_H
5
6 #include "Boss.h"
7
8 class CEO: public Boss
9 {
10 protected:
11     int stock_options;
12 public:
13     void AskEmpInfo();
14     void WriteEmpInfo();
15 };
16
17 #endif
```

Once again we see that the CEO's *Ask* and *Write* functions make calls to its parent class' (i.e., Boss) *Ask* and *Write* functions. It is like a chain reaction when we call the CEO's *Ask* or *Write* function. The CEO functions make calls to the Boss' functions, which in turn calls the Employee's functions. The end result is that each class in the inheritance chain takes care of its own data.

Program 8-3

```
1 //File: CEO.cpp
2
3 #include "CEO.h"
4 #include <iostream>
5 using namespace std;
6
7 void CEO::AskEmpInfo()
8 {
9     Boss::AskEmpInfo();
10    cout << "\n How many shares do I get? ";
11    cin >> stock_options;
12    cin.ignore();
13
14 }
15
16 void CEO::WriteEmpInfo()
17 {
18     Boss::WriteEmpInfo();
19     cout << "\n Shares: " << stock_options;
20 }
21
```



Whew! We're almost finished with the development of the Employee, Boss, and CEO inheritance example. We've written all the classes, now let's see it in action. In the CEODriver.cpp file, we make one CEO object and call its *Ask* and *Write* functions. What happens "behind the scene" is a series of calls to the parent classes' *Ask* and *Write* functions. Figure 8-5 illustrates these calls. Looking at the output, you'll see questions and results from the various class functions.

Program 8-3

```
1 //File: CEODriver.cpp
2 //This program creates one CEO object.
3
4 #include <iostream>
5 using namespace std;
6 #include "CEO.h"
7
8 int main()
9 {
10    cout << "\n\n Work, Work, Work. \n";
11    CEO BigCheese;
```

```

13
14     cout << "\n Ask for info on the CEO ";
15     BigCheese.AskEmpInfo();
16
17     cout << "\n Writing info on the CEO ";
18     BigCheese.WriteEmpInfo();
19
20     cout << "\n\n No more work to do.\n\n";
21     return 0;
22 }

```

Output

Work, Work, Work.

```

Ask for info on the CEO
Enter the employee's name Uncle Scrooge
Enter the employee's SSN 111-22-3333
Enter the dept code A1
Enter the yearly salary 25000

```

What is my bonus this year?\$\$\$\$\$ 50000

How many shares do I get? 2500

```

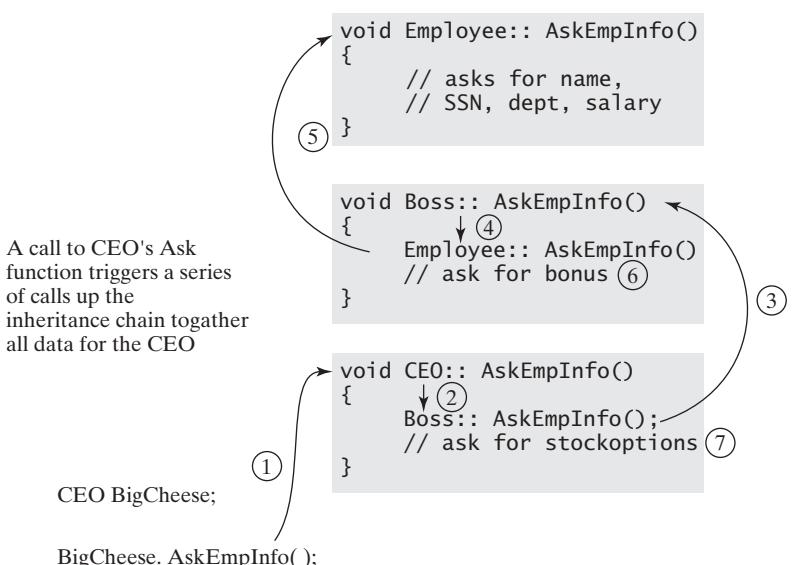
Writing info on the CEO
Employee: Uncle Scrooge
    SSN: 111-22-3333
    Dept: A1
    Salary: $25000
    Bonus: $50000
    Shares: 2500

```

No more work to do.

Figure 8-5

The CEO's *AskForInfo()* function call results in calls to functions up the inheritance chain.



8.3

Access Specifier Specifics and Multiple Inheritance

There are three access specifiers for classes: private, protected, and public. We are familiar with the role these three play when they are used inside a class declaration, such as:

```
class C
{
private:
    // members accessible only to class members
protected:
    // members that are inherited by the child class
    // treated as private to the world
public:
    // members that are inherited by child class are
    // accessible to the world via an object of class C
};
```

Another place we use these specifiers is in the first line of a class declaration. When used in the first line of a declaration, they are known as **base class access specifiers**. The format is shown below:

```
class A : public B
{
    :
    :
};
```

The “public B” is the base class access specifier. It is possible to create derived classes by using all three of the specifiers: public, protected, and private. This base specifier dictates how the base members are treated in the derived classes. Table 8-2 summarizes the base class specifiers if class A (below) is used as a base class.

```
class A
{
private:
    // private members
protected:
    // protected members
public:
    // public members
};
```

Multiple Inheritance

Beginning C++ programmers often ask if it is possible to have **multiple inheritance**, that is, it is possible to derive a new class from two or more base classes.

base class access specifier

an access specifier used in the first line of a declaration dictates how the base members are treated in the derived classes

multiple inheritance

two or more classes used as base classes for a new class

TABLE 8-2Base Class Specifiers in Derived Class Declarations^a

Public	Protected	Private
<pre>class B: public A { };</pre> <p>All public members of class A become public members of class B.^b</p> <p>All protected members of class A become protected members of class B.</p>	<pre>class C: protected A { };</pre> <p>All public and protected members of class A become protected members of class C.</p>	<pre>class D: private A { };</pre> <p>All public and protected members of class A become private members of class D.</p>

^aPrivate members of base class A are not inherited by class B, C, or D. Private class members (data and functions) are never passed to derived classes.

^bConstructors are not inherited by the child class

The answer is yes, it is possible to have multiple parents for a child class. The syntax for this type of inheritance is shown here:

```
class A
{
    // base class
};

class B
{
    // base class
};

class C : public A, public B
{
    // derived class from two base classes
    // class C has both public and protected members from A and B
    :
    :
};
```

The derived class must have the parent classes in a comma-separated list, and base access specifiers are needed. To design a class with multiple base classes, you should be sure that the new class does indeed have an *is a* relationship with both base classes. For example, if you are writing a program for Pets (e.g., cats, dogs, hamsters, etc.) you may derive a class Cats by using both a HousePet class and a Mammal class because a cat is a HousePet and a cat is a Mammal. Beginning C++ programmers may believe that using a multiple-base inherited derived class is a

time-saving solution for their program. In truth, multiple derived classes pose rather complicated design and implementation issues. This programming design should be attempted with extreme caution.

8.4

Inheritance, Constructors, and Destructors

Perhaps by now you have noticed that we have not mentioned constructors (and destructors) and inheritance. There are several rules in C++ concerning base and derived class constructor relationships when constructors are called and when they are not and parameter passing between base and derived constructors. It is important for the new programmer to get a feel for inheritance and base and derived classes before worrying about these rules. The Counter and Employee examples are good places to start.

Review of Constructors and Destructors

A constructor function is a class member function that has the same name as its class. This function is executed when the object is created. The main job of a constructor is to initialize object data to known values and thus avoid unexpected behavior from the object. It is possible to overload class constructors. Overloading class constructors offers the programmer many ways to create new objects.

A destructor function is a class member that also has the same name as its class. This function is executed when the object is destroyed. The destructor function's job is to perform any clean-up activity such as closing files or freeing memory before the object goes out of existence. The destructor function cannot be overloaded.

Base and Derived Class Default Constructor Functions—No Input Parameters

Default constructor functions should be placed in all the classes that you write. These constructors have no input values and whose job is to initialize class data values. When a derived object is created using the no-input, default constructor, its base class constructor function (if it exists) is executed first, followed by the derived class' constructor. If there is a chain of parent classes, then the “root” parent constructor is executed first, then the chain of child constructors are executed in order (as seen in the CEO program).

The base constructor function is executed first, followed by the derived constructor function. Because the derived class depends on the base class, the base class constructors must be executed before the derived constructor. As long as there are no input parameters to either the derived or base class constructor functions, the program will execute the base constructor then the derived constructor. Remember, the default constructor's job is to initialize data to reasonable and safe

values. If you always use the default constructor in your classes, then you are assured the parent data is initialized too.

Passing Parameters into Overloaded Derived Constructor Functions

It is wonderful to build a base class and have a derived class inherit the data and functions of the base class. Programmers often find they create only derived class objects in programs, yet the programmers still need to initialize the base class variables when the derived object comes into existence. Remember, a child class inherits the protected and public members of the class. These base class members are contained in the derived class.

When working with inherited classes and constructors, a programmer should simply allow the derived class constructor function to call the base constructor function and pass the initial values to it. This programming technique is more efficient and typically involves less risk. We'll see an example of this in the next program.

Base and Derived Classes and Destructor Functions

Class destructor functions may also be placed in both base and derived class declarations. You need destructors if the object needs to do any clean up work before it goes out of scope. Clean up work typically involves tasks such as closing data files or freeing allocated memory. In these examples, we write a statement to the screen in order to see that the destructor is being executed. The order of execution for the destructors is opposite that of constructor function execution. When a child class object is destroyed; the derived class destructor is called before the base class destructor function. The language is designed to build (construct) the parent before it builds the child. The opposite is true for destruction—the child is destructed and then the parent.

An example is needed to illustrate the construction/destruction principle. Let's stick with something simple to see how C++ handles default and overloaded constructors in inherited classes.

The Doctor is a Person program

In Program 8-4 we illustrate how C++ calls (or doesn't call) default and overloaded class constructor functions. We build a simple Person class that has data for a Person's name and age. There are two constructors, a *set* function, *write* and *destructor* function. Admittedly, not very exciting to look at, but it gets better as we go along.

Program 8-4

```
1 //File: Person.h
2
3 #ifndef _Person_H
```

```
4 #define _Person_H
5
6 #include <iostream>
7 #include <string>
8 using namespace std;
9
10 class Person
11 {
12 protected:
13     string name;
14     int age;
15 public:
16     Person();
17     Person(string n, int a);
18     void SetData(string n, int a);
19     void WritePerson();
20     ~Person();
21 };
22
23 #endif
```

The associated class function for the Person is seen here:

Program 8-4

```
1 //File: Person.cpp
2
3 #include "Person.h"
4
5 Person::Person()
6 {
7     cout << "\n In the default Person constructor";
8     name = "";
9     age = 0;
10 }
11
12 Person::Person(string n, int a)
13 {
14     cout << "\n In the overloaded Person constructor";
15     name = n;
16     age = a;
17 }
18
19 void Person::SetData(string n, int a)
20 {
21     name = n;
22     age = a;
```

```
23  }
24
25 Person::~Person()
26 {
27     cout << "\n Destructing Person " << name << endl;
28 }
29
30 void Person::WritePerson()
31 {
32     cout << "\n " << name << " is " << age <<
33         " years old. ";
34 }
```

We now build a Doctor class, which is derived from the Person class. After all, a Doctor is a Person. The Doctor has a medical specialty string. Notice how the overloaded constructor and *Set* function are passed all three pieces of data.

Program 8-4

```
1 // File: Doctor.h
2
3 #ifndef _DOCTOR_H
4 #define _DOCTOR_H
5
6 #include "Person.h"
7
8 #include <iostream>
9 using namespace std;
10
11 class Doctor:public Person
12 {
13 private:
14     string specialty;
15
16 public:
17     Doctor();
18     Doctor(string n, int a, string spec);
19     void SetData(string n, int a, string spec);
20
21     void WriteDoctor();
22
23     ~Doctor();
24 };
25
26 #endif
```

Things begin to get exciting when we examine the Doctor's *SetData* and constructor functions. First, the *SetData* function—notice that it is passed the three data items

and it turns around and passes the name and age to the Person's *SetData* function. We did this same trick with the *Ask* and *Write* function in the CEO program.

```
void Doctor::SetData(string n, int a, string spec)
{
    Person::SetData(n,a);      // <-- pass data to Person's SetData
    specialty = spec;
}
```

Now examine the two constructors. Notice how the default constructor sets the specialty value to “”. The Person constructor sets the name to “” and age to zero. C++ calls the default parent constructor automatically—however, the function header line of the Doctor's overloaded constructor is new. It is:

```
Doctor::Doctor(string n, int a, string spec): Person(n, a)
```

This is an explicit call to the Person's overloaded constructor. We're passing the name and age values to it. This is how the code is written in order to pass data from a constructor to its parent. Here is the source code for the Doctor class functions.

Program 8-4

```
1 // File: Doctor.cpp
2
3 #include "Doctor.h"
4 #include <iostream>
5 using namespace std;
6
7 Doctor::Doctor()
8 {
9     cout << "\n In the default Doctor constructor ";
10    specialty = "";
11 }
12
13 Doctor::Doctor(string n, int a, string spec): Person(n, a)
14 {
15     cout << "\n In the overloaded Doctor constructor ";
16     specialty = spec;
17 }
18
19 void Doctor::SetData(string n, int a, string spec)
20 {
21     Person::SetData(n,a);
22     specialty = spec;
23 }
24
25 void Doctor::WriteDoctor()
26 {
27     Person::WritePerson();
```



```

28     cout << "\n Specialty: " << specialty;
29 }
30
31 Doctor::~Doctor()
32 {
33     cout << "\n Destructing Doctor with a specialty of "
34         << specialty;
35 }
36

```

Let's now construct two doctor objects in a *main* function. You guessed it, we're going to build one using the default constructor and set data into the object. The second doctor object is constructed using the overloaded constructor. The important thing to watch as this program runs is the order of the constructor and destructor calls. See how the program calls the Person constructor before the Doctor constructor, and then destructs the Doctor before the Person. Figure 8-6 shows the default construction and Figure 8-7 shows the construction using the input list constructor. Compare these two figures to see how the two different Doctor constructors work.

Program 8-4

```

1 // File: Driver.cpp
2
3 #include "Doctor.h"
4
5 #include <iostream>
6 #include <string>
7
8 using namespace std;
9
10 int main()

```

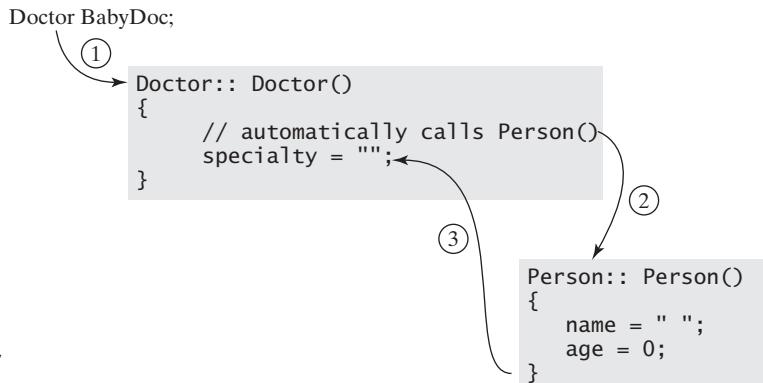
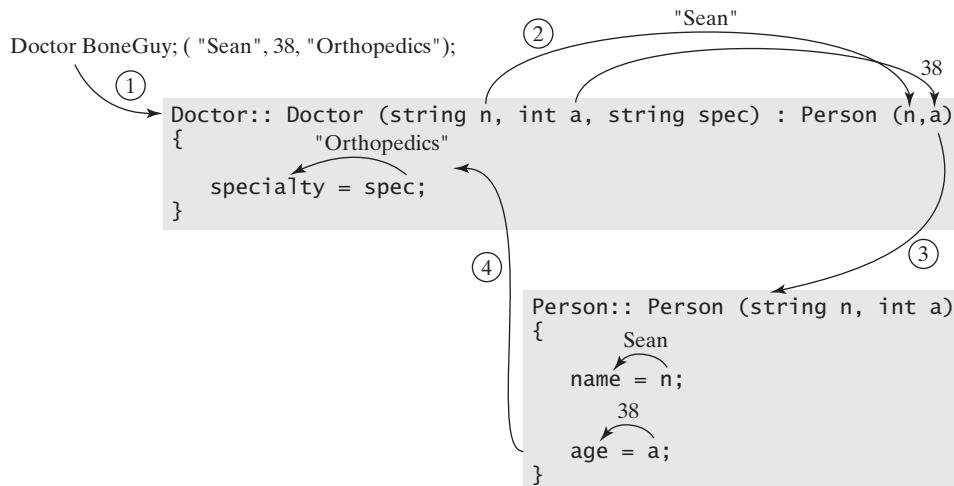


Figure 8-6

The default constructor automatically calls its parent's default constructor.

**Figure 8-7**

The overloaded constructor must explicitly call and pass data to its parent's overloaded constructor.

```

11  {
12      cout << "\n\n Constructors and Inheritance\n";
13
14      cout << "\n Karen is our baby doctor."
15          << "\n Make an object using default "
16          << "constructor, \n then set the data.";
17
18      Doctor BabyDoc;
19      BabyDoc.SetData("Karen", 45, "Deliveries");
20
21      cout << "\n\n Sean is our orthopedic surgeon."
22          << "\n Make an object using the overloaded"
23          << " constructor" ;
24      Doctor BoneGuy("Sean", 38, "Orthopedics");
25
26      cout << "\n\n All finished. End of program.\n";
27      return 0;
28  }

```

Output

Constructors, Destructors and Inheritance

Karen is our baby doctor.
 Make an object using default constructor,
 then set the data.
 In the default Person constructor
 In the default Doctor constructor

```
Sean is our orthopedic surgeon.  
Make an object using the overloaded constructor  
In the overloaded Person constructor  
In the overloaded Doctor constructor  
All finished. End of program.
```

```
Destructing Doctor with a specialty of Orthopedics  
Destructing Person Sean
```

```
Destructing Doctor with a specialty of Deliveries  
Destructing Person Karen.
```

Rules for Derived Class—Base Class Constructors The programmer should keep several points in mind while using a derived class constructor:

1. The base class constructor function is called automatically if there is no derived class constructor—as long as the derived object does not require input parameters.
2. If you need the services only of the base class constructor and there are no inputs to the derived class upon creation, a derived class constructor is not required.
3. If your derived object needs to pass parameters to the base class constructor and there are no specific inputs for the derived class, a derived class constructor must be present. This derived constructor merely acts as a conduit for the parameters.

8.5 Polymorphism and Virtual Functions

Polymorphism—One Interface, Many Forms

polymorphism

one interface, many implementations

Polymorphism is a concept supported by object-oriented languages. A literal definition of **polymorphism** is “one interface, many forms or methods.” What does this definition mean? A computer’s CD drive provides an easy way to illustrate polymorphism. To operate a computer’s CD drive, you press a button on the front of the player and the door glides open. You place a compact disc on the rack and push a button to close the door. The drive starts, whether it is playing music or accessing computer data. No matter who makes CD drives for computers, the drives all operate in the same manner—that is, there is “one interface” (one way to interact with it) and there are “many methods” or “many implementations” (different vendor CD drives).

Another example of a single interface, multiple forms, is a key and lock. There are many different types of lock and key systems, such as the lock on your front door, the lock on your car door, or a padlock. The interface for whatever type of key and lock is the same: you insert the key in the lock and turn the key to unlock the door or padlock. Although the locks have different mechanisms and the keys look different, the key and lock interface is the same.

How do these concepts relate to C++ programming? In C++ programs, it is possible to create several implementations that have the same name, thus reducing program complexity. For example, borrowing from our lock and key idea, it is possible to build several LockandKey classes (i.e., padlock key, door key, car key, etc.). Each class contains a *Lock* and an *Unlock* function specific for that type of lock. If our program then has many different types of *LockandKey* objects, we simply call the *Lock* and *Unlock* functions as needed for each of the different types of objects. Behind the scenes, the correct function is accessed for each object.

You may ask yourself, “Aren’t overloaded functions providing us with multiple implementation with a single function name?” Yes! Overloaded functions and operators are referred to as ***compile-time polymorphism***. When the program is compiled, the compiler and the linker work together so that the location (address) of each function is known for each call. (The computer term for this process is “early binding.”) At program execution time, there is no question about which one of the overloaded functions is to be called. Chapter 4 had four forms of the *SayGoodnight* function. When we compile and link a program that has *SayGoodnight(); SayGoodnight("Bob"); SayGoodnight("Susan", "Melissa");*, and *SayGoodnight(5);*, the program knows exactly which *SayGoodnight* function is called in each call statement once the program is linked.

A second form of polymorphism, known as ***run-time polymorphism***, uses inheritance and virtual functions. The computer term for this is “late binding.” The program sometimes does not know exactly which function is to be called until the program is executing. Why do we care about this? Imagine a base class, such as Shape, that is used to derive many new classes, such as Sphere, Pyramid, Cone, and Box. Assume that each of the Shape children classes have a same-name function, such as *DrawMyself*. If the program user can select one of the Shapes, then the program will call one of the *DrawMyself* functions, but we aren’t sure which shape is drawn until the program executes. This programming method is accomplished by “virtual” functions. These special functions provide a way to access a derived class function via a pointer during run-time. This ability provides a very powerful and flexible programming tool.

compile-time polymorphism

the condition in which the program knows exactly which function will be called at each call statement

run-time polymorphism

the condition in which the program does not know which function will be called until execution

What Is a Virtual Function?

A ***virtual function*** is a class member function located in a base class. When this base class is used to derive a new class, this new child class redefines the virtual function to fit its specific needs. Virtual functions are accessed by objects in the usual fashion—however, the power in virtual functions is achieved when a program uses a pointer to access the functions.

virtual function

function in a base class; the child class redefines this function for its specific use

Virtual Shapes Let’s use an example to show how virtual functions operate. We make a base class called Shape that has a purely virtual function *WhatAmI*. Note that the keyword *virtual* is used in the class member prototype. We derive four classes from Shape: Sphere, Pyramid, Cone, and Box. All child classes have *WhatAmI* functions (no *virtual* keyword). The functions have been written specifically for each shape. When you write a parent class, such as Shape, it is reasonable to expect that the child classes will be similar in general, but different in detail.

Using virtual functions allows the programmer to specify generalized functions for the child classes, and then have a quick and easy way to access them. Let's see our simple example in action.

Program 8-5 has the Shapes.h file, which contains the base class Shape and the four child classes. Notice how the *WhatAmI* virtual function in the Shape class is assigned zero. This indicates that there is no function body for the Shape class. A virtual function does not require that an actual function body exist in the base class. Derived class functions often contain all the code required to perform whatever action is desired. No code is needed in the base class. If a base class virtual function does not have an associated function body, it is said to be a ***pure virtual function***.

pure virtual function

no actual function implementation in the base class

Program 8-5

```
1 //File: Shapes.h
2
3 #ifndef _SHAPE_H
4 #define _SHAPE_H
5
6 #include <iostream>
7 using namespace std;
8
9 class Shape
10 {
11 public:
12     //purely virtual, no actual function exists
13     virtual void WhatAmI() =0 ;
14
15 };
16
17 class Pyramid:public Shape
18 {
19 public:
20     void WhatAmI() { cout << "\n I am a pyramid. \n"; }
21 };
22
23 class Sphere : public Shape
24 {
25 public:
26     void WhatAmI() { cout << "\n I am a sphere. \n"; }
27 };
28
29 class Cone:public Shape
30 {
31 public:
32     void WhatAmI() { cout << "\n I am a cone. \n"; }
33 };
34
```

```
35 class Box : public Shape
36 {
37     public:
38         void WhatAmI() { cout << "\n I am a box. \n"; }
39     };
40
41 #endif
```

If a programmer declares a pointer to the base class and assigns the address of a derived class into that pointer (legal in C++), then the derived class virtual function is accessed via that pointer. In the code below, we declare an array of Shape pointers (Shape *). We then make Sphere, Pyramid, Cone, and Box objects, and assign their addresses into the array. We use the base pointer array and a *for* loop to access all *WhatAmI* functions. C++ looks at the address of the object, and determines which child class function to access. Examine the source code here along with the output. This is a useful and powerful tool for C++ programmers!

Program 8-5

```
1 //File: VirtualShapesDriver.cpp
2
3 #include <iostream>
4 using namespace std;
5 #include "Shapes.h"
6
7 int main()
8 {
9     cout << "\n The Virtual Shapes Program" << endl;
10
11     Shape *basePtrs[4];      //make an array of
12                           //base class pointers
13
14     Sphere MySphere;
15     Pyramid MyPyramid;
16     Cone MyCone;
17     Box MyBox;
18
19     basePtrs[0] = &MyCone;   //place addresses of child
20     basePtrs[1] = &MySphere; //objects into base pointers
21     basePtrs[2] = &MyPyramid;
22     basePtrs[3] = &MyBox;
23
24     //can now loop through array to
25     //access each object's WhatAmI function
26     for(int i = 0; i < 4; ++i)
27     {
28         basePtrs[i]->WhatAmI();
29     }
```



```

30
31     cout << "\n Very cool, don't you think? " << endl;
32     return 0;
33 }
```

Output

The Virtual Shapes Program

I am a cone.
I am a sphere.
I am a pyramid.
I am a box.

Very cool, don't you think?

How Are Virtual Functions Useful?

Game Programming Many different programming scenarios using virtual functions make a programmer's life much easier. Let's pretend you are writing the software for an adventure game based in New Mexico in the 1700s. In this game, there are different characters that may be drawn into the playing window, depending on the conditions in the current scene. These characters include our hero, her band of traveling companions, a dog, a pack of burros, a herd of sheep, and a handsome man—our hero's love interest. At any given time, you expect to have anywhere from one to thirty characters on the screen.

There are several design approaches for this game. One approach involves setting up a base class for all the characters in the game. We place several virtual functions in this base class, including AmIOnScreen (keeps track of whether the character is currently drawn on the screen), WhereAmI (reports screen location), and Draw (contains drawing information for the character). The base class and two children classes are shown here:

```

class GameCharacter
{
private:
    Screen Location;           // Screen is class for coordinates
                                // other general data
public:
    virtual Screen WhereAmI(); 
    virtual bool AmIOnScreen();
    virtual void Draw();
};

class Hero : public GameCharacter
{
    // Hero data
public:
    Screen WhereAmI();
    bool AmIOnScreen();
    void Draw();
};
```

```
class Sheep: public GameCharacter
{
    // Sheep data
public:
    Screen WhereAmI();
    bool AmIOnScreen();
    void Draw();
};
```

The main part of the game declares not only a hero object but all the supporting characters listed above as well. A graphics game such as this one probably will be built in the Windows environment (if the target platform is a personal computer). The graphics may be either OpenGL or DirectX. For our example, we simply pretend there is a *main* function:

```
int main()
{
    Hero Rosita;           //All game character objects declared here.
    Sheep BaaBaa[5];
    Dog Perro;
    //etc.
```

In the screen-handling portion of the game, we need to create an array of GameCharacter pointers and assign the addresses of all the character objects.

```
GameCharacters *ptr_GameChar[100];
int NumberCharacters;
ptr_GameChar[0] = &Rosita;      //first pointer holds the hero's address
ptr_GameChar[1] = &Perro;       //next pointer holds the dog's address
for(i = 0; i < NumSheep; ++i) //fill the sheep's addresses
{
    ptr_GameChar[i+2] = &BaaBaa[i];
}
```

Once the GameCharacter pointer array contains the addresses of all the character objects, the drawing portion of the game can be handled in a simple *for* loop.

```
Screen place;           //local variable to hold screen coords
for (i = 0; i < NumberCharacters; ++i)
{
    //if true this char is on the screen
    if(ptr_GameChar[i]->AmIOnScreen())
    {
        //each object knows where it is
        place = ptr_GameChar[i]->WhereAmI();
        // use the screen coords and check for placement
        ptr_GameChar[i]->Draw();    //calls correct drawing routine
    }
}
```



Each game character object knows its location, whether it is on the screen at the current time and how to draw itself. The programmer does not need to use large *switch* statements to determine which routine must be called; virtual functions handle that detail. There is always a small amount of overhead associated with setting up pointer arrays and filling them with addresses, but the payoff is great!

One Last Example: Two Vehicles Let's do one last program to illustrate inheritance, array of pointers to base classes, and virtual functions. The Two Vehicles program contains a base class for vehicle information. From this class, we derive two different vehicle classes: recreational (RV) and commercial (Semi). The base class has strings for owner, license, and vehicle identification number (VIN). The member functions include virtual functions for obtaining and writing the information. To keep our example simple, we have empty constructors in all three classes. The base class virtual function can contain code, and it must be explicitly called in the child function. We've also placed the three classes in one file. Look over this Vehicle.h file.

Program 8-6

```
1 //File: Vehicles.h
2 //To keep it simple, we'll leave all
3 //three class declarations in this file.
4
5 //If the classes were of any size, they
6 //should be placed in separate .h files.
7
8 #ifndef _VEHICLES_H
9 #define _VEHICLES_H
10
11 #include <iostream>
12 #include <string>
13 using namespace std;
14
15
16 class Vehicle
17 {
18 protected:
19     string owner, license;
20     string VIN;
21
22 public:
23     Vehicle();
24     virtual void AskInfo();
25     virtual void WriteInfo();
26 };
27
28
```

```
29 class RV:public Vehicle
30 {
31     private:
32         char category; //RV class A, B, C
33
34     public:
35         RV();
36         void AskInfo();
37         void WriteInfo();
38     };
39
40
41 class Semi:public Vehicle
42 {
43     private:
44         double weightCap;      //weight capacity
45     public:
46         Semi();
47         void AskInfo();
48         void WriteInfo();
49     };
50
51 #endif
```

Now examine the Vehicle.cpp file. It contains the function code for our three vehicle classes. Notice how the child classes make calls to the parent's functions.

Program 8-6

```
1 //File: Vehicles.cpp
2
3 #include <iostream>
4 using namespace std;
5 #include "Vehicles.h"
6
7 void Vehicle::AskInfo()
8 {
9     cout << "\n Enter owner's name: ";
10    getline(cin, owner);
11
12    cout << "\n Enter license plate such as NM JAM 099 : ";
13    getline(cin, license);
14
15    cout << "\n Enter VIN : ";
16    getline(cin, VIN);
17 }
18
```

```

19 void Vehicle::WriteInfo()
20 {
21     cout << "\n\n    Owner: " << owner
22             << "\n    License: " << license
23             << "\n        VIN: " << VIN << endl;
24 }
25
26 void RV::AskInfo()
27 {
28     cout << "\n Enter info for the RV.";
29
30     Vehicle::AskInfo(); //call the base class AskInfo first
31
32     cout << "\nEnter RV class A, B, or C ";
33     cin >> category;
34     cin.ignore(); // pull off enter key left by cin
35 }
36
37 void RV::WriteInfo()
38 {
39     Vehicle::WriteInfo();
40     cout << "\n This RV is a class " << category;
41 }
42
43 void Semi::AskInfo()
44 {
45     cout << "\n Enter info for the Semi Truck.";
46     Vehicle::AskInfo();
47
48     cout << "\nEnter the weight capacity ";
49     cin >> weightCap;
50     cin.ignore();
51 }
52
53 void Semi::WriteInfo()
54 {
55     Vehicle::WriteInfo();
56     cout << "\n This commercial vehicle has a "
57             << " weight capacity of " << weightCap
58             << " pounds." << endl;
59 }
```

The TwoVehiclesDriver.cpp file contains the *main* function. Once again, we set up an array of base class pointers, create two child classes and assign their addresses into the array elements. We then use *for* loops to access the virtual functions defined in the classes.

Program 8-6

```

1 //File: TwoVehiclesDriver.cpp
2
```

```
3 #include <iostream>
4 #include "Vehicles.h"
5
6 using namespace std;
7
8 int main()
9 {
10     cout << "\n The Two Vehicles Program \n " << endl;
11
12     Vehicle *base[2]; //2 pointers to the base class
13     RV HaveFun;
14     Semi GoToWork;
15
16     base[0] = &HaveFun; //assign into base class ptrs
17     base[1] = &GoToWork;
18
19     //obtain the info using the virtual functions
20     for(int i = 0; i < 2; ++i)
21     {
22         base[i]->AskInfo();
23     }
24
25     //now write the information
26     for(i = 0; i < 2; ++i)
27     {
28         base[i]->WriteInfo();
29     }
30
31     cout << "\n\n Work? No. Let's go camping! \n\n";
32     return 0;
33 }
```

Output

The Two Vehicles Program

```
Enter info for the RV.
Enter owner's name: Jennifer Marie
Enter license plate such as NM JAM 099 : GA 877FUN
Enter VIN : 984KEN3563
Enter RV class A, B, or C      B

Enter info for the Semi Truck.
Enter owner's name: Hannah Marie
Enter license plate such as NM JAM 099 : NM HLR825
Enter VIN : 843WHD6673
Enter the weight capacity 20000
```

```
    Owner: Jennifer Marie
    License: GA 877FUN
    VIN: 984KEN3563
This RV is a class B
```

Owner: Hannah Marie

License: NM HLR825

VIN: 843WHD6673

This commercial vehicle has a weight capacity of 20000 pounds.

Work? No. Let's go camping!

8.6 Summary

Inheritance is an important principle in object-oriented software. It involves the ability to take an existing class and derive new classes from it. This ability to create new classes from existing ones opens the door to writing software that is modular and expandable. It allows the programmer to customize and/or modify code through the use of derived classes, with minimal modifications to the base classes.

commercial off-the-shelf (COTS) software

products that may be purchased by software developers to use in the developer's own software

Also, many commercial software businesses build **commercial off-the-shelf (COTS) software** libraries that may be incorporated into software projects. COTS software often provides complete tools with specialized software functionality. Software developers often use a COTS product whenever possible because it is usually much cheaper to buy a package than to write code from scratch. For example, if you are writing a program that requires two-dimensional x-y plots, several COTS graphing products are relatively easy to hook into the code. (Figuring out the way to call the functions is much easier than writing your own two-dimensional graphics package.) Many of the tools available today are object-oriented and provide the developer with classes that can be used either directly or as base classes for custom work. Other COTS software products include translator software that reads and writes certain types of data files (such as image files or computer-aided design files). If your software needs to support industry standard file formats, chances are excellent that translator packages are available. It is not uncommon for translator software developers to add custom features on request—for a fee, of course.

One last note before we leave the subject of inheritance, remember to keep it simple, sweetie. Classes and inheritance are powerful software tools that may easily be misused and made overly complex. It is possible to write C++ code that contains derived classes many layers deep. After the first two or three layers, a programmer may forget what data and functions an object contains. Design your software in a straightforward manner and document your source code. Limit your class derivations and you will be a much happier software developer. (And the person who has to maintain your code in the future will be happy, too!)

8.7 Practice!

Pay Calculations and Virtual Functions

Our first practice program (Program 8-7) makes use of virtual functions to calculate weekly pay for staff members who work at the C++ Auto Dealership. It is common practice for retail businesses to have regular-paid staff employees whose salary is based on the number of hours worked per week times an hourly rate, with

overtime pay for hours over 40. Sales personnel's salary is based on a commission (percentage) of the total dollar amount of the items sold. For this program, we create a CPP_Employee class, and then derive two classes from it. We use purely virtual functions for the *CalculatePay* and *Write* functions.

Our code is designed so that the individual classes are organized into their *.h and *.cpp files. Our exception is that our CPP_Employee class can be completely defined in its *.h file.

Program 8-7

```
1 //File: CPP_Employee.h
2
3 // This class represents all individuals
4 // who work at the C++ Auto Dealership
5
6 #ifndef _CPP_EMP_H
7 #define _CPP_EMP_H
8
9 #include <string>
10 #include <iostream>
11 using namespace std;
12
13 class CPP_Employee
14 {
15 protected:
16     string name;
17     float weeklyPay;
18 public:
19     CPP_Employee(){ name = "", weeklyPay = 0;}
20
21     //These virtual functions must be
22     //defined in the child classes.
23     virtual void CalculatePay() = 0;
24     virtual void WritePayInfo() = 0;
25
26     //We write the Ask function here, as it
27     //only asks for the Employee's name.
28     virtual void AskForPayInfo()
29     {
30         cout << "\n Please enter the employee's name: ";
31         getline(cin,name);
32     }
33 };
34
35 #endif
```

The CPP_Staff and CPP_Sales classes contain data pertinent to their specific type of employee. Notice how the *CalculatePay* function is called from main, too. You could argue that there is no need to burden the object with calling the

Calculate function once the data has been obtained. For this example, we'll use our virtual function and base class pointer to access it.

Program 8-7

```
1 //File: CPP_Staff.h
2
3 // This class represents the admin staff
4 // who work at the C++ Auto Dealership
5
6 #ifndef _CPP_STAFF_H
7 #define _CPP_STAFF_H
8
9 #include "CPP_Employee.h"
10 #include <string>
11 using namespace std;
12
13 class CPP_Staff: public CPP_Employee
14 {
15 private:
16     float hrsWorked, hrRate;
17
18 public:
19     CPP_Staff();
20
21     //define these for a staff person
22     void CalculatePay();
23     void AskForPayInfo();
24     void WritePayInfo();
25 };
26
27 #endif
```

Program 8-7

```
1 //File: CPP_Staff.cpp
2 #include "CPP_Staff.h"
3 #include <iostream>
4 using namespace std;
5
6 CPP_Staff::CPP_Staff()
7 {
8     hrsWorked = 0;
9     hrRate = 0;
10 }
11
12 void CPP_Staff::CalculatePay()
```

```
13  {
14      //1.5 * rate for all hours over 40.0
15      if(hrsWorked <= 40.0)
16          weeklyPay = hrsWorked * hrRate;
17      else
18      {
19          weeklyPay = 40.0 * hrRate
20          + (hrsWorked - 40.0) * 1.5 * hrRate;
21      }
22  }
23
24 void CPP_Staff::AskForPayInfo()
25 {
26     CPP_Employee::AskForPayInfo();
27
28     cout << "\n Enter hourly rate $ ";
29     cin >> hrRate;
30
31     cout << "\n Enter hours worked this week ";
32     cin >> hrsWorked;
33     cin.ignore();
34 }
35
36 void CPP_Staff::WritePayInfo()
37 {
38     cout.precision(2);
39     cout.setf(ios::fixed);
40     cout << "\n Staff person: " << name
41         << "\n Weekly Pay $" << weeklyPay << endl;
42 }
43
```

Program 8-7

```
1  //File: CPP_Sales.h
2
3  // This class represents the sales staff
4  // who work at the C++ Auto Dealership
5
6  #ifndef _CPP_SALES_H
7  #define _CPP_SALES_H
8
9  #include "CPP_Employee.h"
10 #include <string>
11 using namespace std;
12
13 class CPP_Sales: public CPP_Employee
14 {
```

```
15  private:  
16      float commission, weeklySales;  
17  
18  public:  
19      CPP_Sales();  
20  
21      //define these for a sales person  
22      void CalculatePay();  
23      void AskForPayInfo();  
24      void WritePayInfo();  
25  };  
26  
27 #endif
```

Program 8-7

```
1 //File: CPP_Sales.cpp  
2 #include "CPP_Sales.h"  
3 #include <iostream>  
4 using namespace std;  
5  
6 CPP_Sales::CPP_Sales()  
7 {  
8     commission = 0;  
9     weeklySales = 0;  
10 }  
11  
12 void CPP_Sales::CalculatePay()  
13 {  
14     //Sales employees are paid on commission.  
15     weeklyPay = weeklySales * commission/100.0;  
16 }  
17  
18 void CPP_Sales::AskForPayInfo()  
19 {  
20     CPP_Employee::AskForPayInfo();  
21  
22     cout << "\n Enter commission rate in %, i.e. 15 for 15% ";  
23     cin >> commission;  
24  
25     cout << "\n Enter sales for the week ";  
26     cin >> weeklySales;  
27     cin.ignore();  
28 }  
29  
30 void CPP_Sales::WritePayInfo()  
31 {  
32     cout.precision(2);  
33     cout.setf(ios::fixed);
```

```
34     cout << "\n Salesperson: " << name
35         << "\n Weekly Pay $" << weeklyPay << endl;
36 }
```

The PayCheckDriver.cpp file contains the *main* function. We build the base class pointer array, assign our objects' addresses into the elements and loop through the array. We are able to access the various objects in an efficient manner.

Program 8-7

```
1 // File: PayCheckDriver.cpp
2 // This program uses virtual functions to
3 // calculate weekly pay for three
4 // C++ Auto Dealership employees.
5
6 #include <iostream>
7 using namespace std;
8
9 #include "CPP_Staff.h"
10 #include "CPP_Sales.h"
11
12 int main()
13 {
14     cout << "\n Pay day at the C++ Car Dealership \n"
15         << "\n Calculate pay for one staff member"
16         << " and two salespersons. \n";
17
18     //one secretary, two salesmen
19     CPP_Staff admin;
20     CPP_Sales seller1, seller2;
21
22     //make an array of base class pointers
23     CPP_Employee *emp[3];
24
25     //load array with sales and staff addresses
26     emp[0] = &admin;
27     emp[1] = &seller1;
28     emp[2] = &seller2;
29
30     //Loop through all employees
31     //Ask for and calculate pay
32     for(int i = 0; i < 3; ++i)
33     {
34         emp[i]->AskForPayInfo();
35         emp[i]->CalculatePay();
36     }
37
38     //Now write everyone's pay
```

```
39      for( i = 0; i < 3; ++i)
40      {
41          emp[i]->WritePayInfo();
42      }
43
44      cout << "\n Sweet!\n";
45      return 0;
46  }
```

Output

Pay day at the C++ Car Dealership

Calculate pay for one staff member and two salespersons.

Please enter the employee's name: Gail M

Enter hourly rate \$ 14.50

Enter hours worked this week 45

Please enter the employee's name: Ron B

Enter commission rate in %, i.e. 15 for 15% 2

Enter sales for the week 75500

Please enter the employee's name: Mary C

Enter commission rate in %, i.e. 15 for 15% 2.5

Enter sales for the week 68525

Staff person: Gail M

Weekly Pay \$688.75

Salesperson: Ron B

Weekly Pay \$1510.00

Salesperson: Mary C

Weekly Pay \$1713.13

Sweet!

DayDate, A New Type of Date Class

Do you remember the first class we wrote in Chapter 7? It was the Date class. Turn back to page 373 and review how we created a class that had month, day, year, description, day in the year, and a flag for leap year. The one piece of information we don't have in our Date class is the day of the week for our date. Let's take advantage of the time functions that provide the date and time, and have it give us the day of the week, too.

Instead of modifying a perfectly good Date class, we will use it as a base class and derive a DayDate class. The DayDate will determine the day of the week for our date, assuming that we obtain the date from the system date. (We don't know how to take an arbitrary date and determine the day of the week for it. That would require a perpetual calendar program.) The first order of business is to change the private data to protected in the Date class so that the data is inherited by our new class. (To create this program, the Date.h and Date.cpp files

are copied and moved into our new CH8_DayDate project.) Here is the only change we make to Date.h:

```
class Date
{
    protected:           <== protected! not private
        int month, day, year;
        string description;
        int dayOfYear;
        bool bLeap;
        void CalcDayOfYear();
        void DetermineLeapYear();
```

Our DayDate class contains one new data item, a string for the *dayOfWeek*. We add a private *FigureDayOfWeek* function. It is private because it will be called from the default constructor. Look over our DayDate class here.

Program 8-8

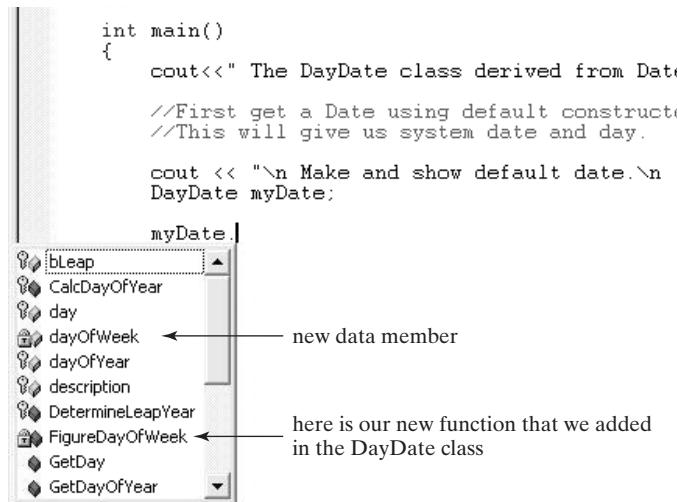
```
1 //DayDate.h
2 //This new DayDate class includes the day
3 //of week in its formatted date string.
4
5 #include "Date.h"
6
7 #ifndef _DAYDATE_H
8 #define _DAYDATE_H
9
10 #include <iostream>
11 using namespace std;
12
13 class DayDate : public Date
14 {
15     private:
16         //can only determine day if using
17         //system date
18         string dayOfWeek;          //Monday, Tuesday, etc
19         void FigureDayOfWeek();
20     public:
21         DayDate();
22         DayDate(int m, int d, int y, string desc);
23         string GetFormattedDate();
24     };
25
26 #endif
```

The DayDate.cpp file contains the constructors, *GetFormattedDate* and *FigureDayOfWeek* functions. Recall that the default constructors are called automatically, so we know that the date information is set into the class members when the *FigureDayOfWeek* function is called. Also, examine the *GetFormattedDate*

function. We call the Date's *Get* function and search the string that we've received for the colon. We then insert the day of week into that string. There is no need for us to re-invent the wheel in this program. Our Date class is well-written, so we can use it wherever possible.

Program 8-8

```
1  //File: DayDate.cpp
2
3  #include "DayDate.h"
4  #include <ctime>
5  using namespace std;
6
7  DayDate::DayDate()
8  {
9      //default Date gets system date
10     FigureDayOfWeek();
11 }
12
13 DayDate::DayDate(int m, int d, int y, string desc):
14 Date(m,d,y,desc)
15 {
16     //can't determine day of week, so we
17     //set this to ""
18     dayOfWeek = "";
19 }
20
21 string DayDate::GetFormattedDate()
22 {
23     //Get string from parent
24     string s = Date::GetFormattedDate();
25
26     //This is in the form:
27     //Description: Month XX, XXXX
28     //So we'll insert the day of week after the:
29
30     int colon = s.find(":");
31
32     s.insert(colon+2, dayOfWeek);
33     return s;
34 }
35
36 void DayDate::FigureDayOfWeek()
37 {
38     string weekdays[7] = {"Sunday ", "Monday ",
39                         "Tuesday ", "Wednesday ", "Thursday ",
40                         "Friday ", "Saturday "};
41 }
```

**Figure 8-8**

The pick list for the *DayDate* object shows the inherited members as well as new class members.

```

42     //The time() gives us day of week number
43     time_t rawtime;
44     tm *OStime;
45
46     time(&rawtime);
47     OStime = localtime(&rawtime);
48
49     int dayNum = OStime->tm_wday;
50     dayOfWeek = weekdays[dayNum];
51 }
52

```

The *main* function is short too. We make a *DayDate* and ask for its formatted date string. We then make a date using our own parameters, and ask it for that formatted date. As you can see the day of the week is in the default date's string. Before we leave this problem, look at Figure 8-8. It shows the pick list presented by Visual C++ when we use the *DayDate*'s object and dot operator. Our *DayDate* inherits the data and functions from our *Date* class.

Program 8-8

```

1  //File: DateDriver.cpp
2
3  #include <iostream>
4  #include "DayDate.h"

```

```
5
6  using namespace std;
7
8  int main()
9  {
10    cout<<" The DayDate class derived from Date. \n";
11
12    //First get a Date using default constructors.
13    //This will give us system date and day.
14
15    cout << "\n Make and show default date.\n ";
16    DayDate myDate;
17
18    //Now ask the DayDate object for its formatted string.
19    cout << myDate.GetFormattedDate() << endl;
20
21    //Now make a date
22    cout << "\n Make and show our own date.\n ";
23    DayDate theDate(3,14, 1998,"Jason's Birthday");
24    cout << theDate.GetFormattedDate() << endl;
25
26    return 0;
27 }
```

Output

The DayDate class derived from Date.

Make and show default date.

Today's Date: Saturday October 7, 2006

Make and show our own date.

Jason's Birthday: March 14, 1998.

Rain Barrel Estimator Program and Inheritance

Our last program in the Practice section of this chapter involves writing a class that has two class objects that we write. We will have one class used as a base class for a new child class. It's simple on the outside but exciting on the inside! First, some background information.

In the desert southwest of the United States, we've had a multi-year drought. As a result we've become very conscious of our water use. Many southwesterners, who enjoy gardening, have started installing rain barrel systems to catch and hold water that flows off their rooftops. Some barrel systems are just 55-gallon drums placed at the end of the down-spout while other systems are more elaborate. Home builders in the mountainous high desert bury large tanks underground and equip the tank with irrigation pumps. The water can then be pumped from the tank and used to irrigate outdoor plants.

In this program, we build a RainBarrelEstimator class, which asks the user for the square footage of the collection surface, such as the flat roof on the house, and an estimate of total precipitation over a period of time. The goal of this program is to determine the total gallons of water that could be captured from this area and stored in the barrel system. To help with the calculations, we first build a RainVolCalc class, which is a rain volume calculator. Given the square footage area and depth of water, it determines the total gallons. (We've performed this sort of calculation in previous programs.)

The fun part of this program is using the RainVolCalc class as a base class and deriving the snow volume calculator class, SnowVolCalc. If you search the Web for snow, you'll find a wealth of information! Did you know that freshly fallen snow referred to as "powder" has a water density of approximately 12%? This means that if you have one inch of new powder, and you melt it to water, you will have 0.12 inches of water. Snow that is packed and "wet" has a water density of around 33% whereas sleet snow that is icy and slushy is 50% water density. Our snow volume calculator needs to know what type of snow is in the collection surface in order to determine an "accurate" volume for the barrels.

Let's begin Program 8-9 by looking at the RainVolCalc class. Because we have written similar programs, the class is straightforward. To keep things simple, these classes have *set* and *get* functions instead of *Ask* and *Write* functions. Notice how that data is protected (not private) in this first class.

Program 8-9

```
1 //File: RainVolCalc.h
2 //This class calculates the number of
3 //gallons of water that covers a given
4 //number of square feet.
5
6 #ifndef _RAIN_VC_H
7 #define _RAIN_VC_H
8
9 #include <string>
10 using namespace std;
11
12 class RainVolCalc
13 {
14 protected:
15     float inches;    //depth of rain
16     int sqFeet;
17     float gallons;
18     void CalculateGallons();
19 public:
20     RainVolCalc();
21     void SetInches(float i){inches = i; }
22     void SetSqFeet(int sf){sqFeet = sf; }
23     float GetGallons();
24     string GetVolumeString();
25 };
```

```
26  
27 #endif
```

The *GetGallons* and *GetStringVolume* functions both call the *CalculateGallons* function before returning the gallons values.

Program 8-9

```
1 //File: RainVolCalc.cpp  
2  
3 #include "RainVolCalc.h"  
4 #include <iostream>  
5 #include <string>  
6 using namespace std;  
7  
8 RainVolCalc::RainVolCalc()  
9 {  
10     sqFeet = 0;  
11     inches = 0;  
12 }  
13  
14 void RainVolCalc::CalculateGallons()  
15 {  
16     //calculate sq inches coverage  
17     int sqInches = sqFeet * 144;  
18  
19     //mult by depth to get cubic in vol  
20     int cubicIn = inches * sqInches;  
21  
22     //231 cubic in = 1 gallon  
23     gallons = cubicIn/231.0;  
24 }  
25  
26 float RainVolCalc::GetGallons()  
27 {  
28     CalculateGallons();  
29     return gallons;  
30 }  
31  
32 string RainVolCalc::GetVolumeString()  
33 {  
34     CalculateGallons();  
35     stringstream ss;  
36     ss << " " << inches << " \" of rain over "  
37     << sqFeet << " sq feet is " << gallons  
38     << " gallons. ";  
39  
40     return ss.str();  
41 }
```

The SnowVolCalc is derived from RainVolCalc class. Determining the volume of water for snow is the same calculation as for rain with one exception. We calculate the water volume for the given depth (the snow class calls the rain's *CalculateVolume* function) and then this volume is reduced according to the water density value of the snow. Study the snow class and its functions and see how they make use of the rain class' functions whenever possible.

Program 8-9

```
1  /*File: SnowStorm.h
2
3  Snow types:
4  powder, freshly, uncompacted snow
5  packed, coarse, granular wet snow
6  sleet, ice pellets formed when snowflakes
7      thaw, then refreeze
8
9  water densities (equivalents values of snow)
10 new powder 12% water
11 1 inch powder = 0.12 inches of rain
12
13 packed snow 33% water
14 1 inch packed = 0.33 inches of water
15
16 sleet or late spring snow pack 50%
17 1 in sleet = 0.50 inches of water
18
19 */
20
21 #ifndef _SNOW_VC_H
22 #define _SNOW_VC_H
23
24 #include "RainVolCalc.h"
25
26 class SnowVolCalc : public RainVolCalc
27 {
28 private:
29     string type;      //powder, packed or sleet
30     double density;   // % of water
31
32     void CalculateGallons(); //have to know what type
33 public:
34     SnowVolCalc();
35     void SetType(string t);
36     string GetVolumeString();
37     float GetGallons();
38 };
39
40 #endif
```

Program 8-9

```
1 //File: SnowVolCalc.cpp
2
3 #include "SnowVolCalc.h"
4 #include <iostream>
5 using namespace std;
6
7 SnowVolCalc::SnowVolCalc()
8 {
9     //default to powder
10    type = "powder";
11    density = 0.12;
12 }
13
14 string SnowVolCalc::GetVolumeString()
15 {
16     //This calls SnowVolCalc's
17     CalculateGallons();
18
19     stringstream ss;
20     ss << " " << inches << " \\" of " << type
21         << " over " << sqFeet
22         << " sq feet yields " << gallons
23         << " gallons. ";
24
25     return ss.str();
26 }
27
28 void SnowVolCalc::CalculateGallons()
29 {
30     //this calcs gallons of rain
31     RainVolCalc::CalculateGallons();
32
33     //reduces gallons by type of snow
34     gallons = gallons * density;
35 }
36
37 float SnowVolCalc::GetGallons()
38 {
39     //this calls SnowVolCalc's function
40     CalculateGallons();
41
42     return gallons;
43 }
44
45 void SnowVolCalc::SetType(string t)
46 {
47     type = t;
```



```
48     //set density value according to type
49     //if unknown type, set to powder
50     if(type == "powder")
51         density = 0.12;
52     else if(type == "packed")
53         density = 0.33;
54     else if(type == "sleet")
55         density = 0.50;
56     else
57         density = 0.12;
58 }


---


```

The RainBarrelEst class provides the framework for the two calculator classes. The estimator class has the code for gathering the user's data, passing it into the appropriate calculator object, obtaining the data from the objects and writing it to the screen. We keep it simple by coding *Ask* and *Write* functions in the estimator. More complicated interfaces could be written for this class, but today we take the shortest and easiest route. Study the RainBarrelEst class declaration and associated class functions. Notice how the data is set into the objects in the *Ask* function and retrieved in the *Write*.

Program 8-9

```
1  //File: RainBarrelEst.h
2
3
4  #ifndef _RB_EST_H
5  #define _RB_EST_H
6
7  #include "RainVolCalc.h"
8  #include "SnowVolCalc.h"
9
10 #include <string>
11 using namespace std;
12
13 class RainBarrelEst
14 {
15     private:
16         RainVolCalc rainy;
17         SnowVolCalc snowy;
18         float sqFootage, rainInches, snowInches;
19         string desc, typeOfSnow, timeframe;
20
21         float barrelCapacity;
22
23     public:
24         RainBarrelEst();
```

```
26      void AskForData();
27      void WriteData();
28
29  };
30
31 #endif
```

Program 8-9

```
1 //File: RainBarrelEst.cpp
2
3 #include "RainBarrelEst.h"
4
5 #include <iostream>
6 #include <string>
7 using namespace std;
8
9 RainBarrelEst::RainBarrelEst()
10 {
11     cout << "\n Hello from your"
12         << " Rain Barrel Estimator Object!"
13         << "\n I'll calculate required barrel"
14         << " capacity given \n rain and snow totals"
15         << " in inches " << endl;
16
17     barrelCapacity = 0;
18 }
19
20 void RainBarrelEst::AskForData()
21 {
22     cout << "\n Description of collection surface? ";
23     getline(cin,desc);
24
25     cout << "\n Timeframe for collection? ";
26     getline(cin,timeframe);
27
28     cout << "\n Enter total square footage"
29         << " of your collection surface: ";
30     cin >> sqFootage;
31
32     cout << "\n Enter total rainfall in inches: ";
33     cin >> rainInches;
34
35     //set rain values into RainVolCalc object
36     rainy.SetInches(rainInches);
37     rainy.SetSqFeet(sqFootage);
38
39     //now ask about snow?
```

```
40     char snowResponse;
41     cout << "\n Will there be any snow? y/n ";
42     cin >> snowResponse;
43     cin.ignore();
44
45     if(snowResponse == 'y')
46     {
47         cout << "\n What type of snow?"
48             << " powder, packed, sleet? ";
49         getline(cin,typeOfSnow);
50
51         cout << "\n How many inches of "
52             << typeOfSnow << " will be on your "
53             << desc << "? ";
54         cin >> snowInches;
55         cin.ignore();
56
57         snowy.SetType(typeOfSnow);
58         snowy.SetInches(snowInches);
59         snowy.SetSqFeet(sqFootage);
60     }
61 }
62
63 void RainBarrelEst::WriteData()
64 {
65     cout << "\n Results for the " << desc
66         << "\n during " << timeframe << endl;
67
68     cout << rainy.GetVolumeString() << endl
69         << snowy.GetVolumeString() << endl;
70
71     float totalGal = rainy.GetGallons() +
72                     snowy.GetGallons();
73
74     cout << "\n Total Gallons for Barrel: "
75         << totalGal << endl;
76 }
```

The *main* function for this program is short and sweet. We make a barrel estimator object and call the *Ask* and *Write* functions. Study the output from this program following the source code.

Program 8-9

```
1 //File: RainBarrelDriver.cpp
2
3 #include <iostream>
4 #include <string>
5 using namespace std;
```

```
6
7  #include "RainBarrelEst.h"
8
9  int main()
10 {
11     RainBarrelEst barrel;
12
13     barrel.AskForData();
14     barrel.WriteData();
15
16     return 0;
17 }
```

Output

Hello from your Rain Barrel Estimator Object!

I'll calculate required barrel capacity given
rain and snow totals in inches

Description of collection surface? Horse Barn Roof

Timeframe for collection? December 2006 through March 2007

Enter total square footage of your collection surface: 800

Enter total rainfall in inches: 3

Will there be any snow ? y/n y

What type of snow? powder, packed, sleet? powder

How many inches of powder will be on your Horse Barn Roof? 20

Results for the Horse Barn Roof

during December 2006 through March 2007

3 " of rain over 800 sq feet is 1496.1 gallons.

20 " of powder over 800 sq feet yields 1196.88 gallons.

Total Gallons for Barrel: 2692.99

REVIEW QUESTIONS AND PROBLEMS**Short Answer**

1. Is it possible for derived classes to access the private data in their base class?
2. What must happen to allow private members of a class to be inherited by a derived class?
3. If a child class is to contain the public and protected members of its parent class as protected members, which access specifier must be used in the line:
class ClassName : ??? ParentClassName?
4. When is an empty base class constructor function necessary?
5. What is meant by a purely virtual function?
6. Describe the order of function execution for base and derived constructor functions and for destructor functions.

7. Locks and keys and CD players are examples of polymorphism: “one interface, many methods.” Can you think of another example?
8. Name three goals for well-written software.
9. Describe how a pure virtual function (the prototype) is written.
10. Search the Internet for two COTS packages that provide C++ classes.
11. How can a child class function make a call to its parent’s class function?
12. If the overloaded constructor is used when constructing a child class object, but the child’s constructor doesn’t explicitly call the parent’s overloaded constructor, which, if any parent constructor is called?
13. We saw the `CDialog` class as an example of a graphical user interface control that is designed to be used as a parent/base class. Can you find two more GUI classes designed with the same intention? (Note, many GUI controls are used as is, such as a slider or a button. This question is asking for a control that you modify for your program’s use.)
14. If you have a new derived class, will C++ automatically initialize the class members to “zero” values? Explain.
15. Is a protected class member public or private to the world? Is it available to all class functions? Explain.
16. Why does it make sense to constructor the parent object before the child object? Explain.
17. Why does it make sense to destruct the child object before the parent object? Explain.
18. If a base class has public `Set` and `Get` functions for its data members, does a child of that class inherit these Sets and Gets or does the programmer have to write new `Set` and `Get` functions? Explain.
19. If you declared an object that contained a virtual function, is it possible to call this function directly with an object (instead of with a base-class pointer)?
20. If you write a class that contains purely virtual functions, and then you write a child class but don’t write the virtual function, what happens when you compile your program? What happens when you try and run it?

Programming Problems

Write complete C++ programs for the following problems.

21. Write a program that sets up a base class `Sailor` containing protected data for name, rank, and serial number. There should be two constructor functions: one is empty and the other has the three data values as inputs. The other member functions should include an `AskInfo` function (asks the user for Sailor information) as well as a `WriteInfo` (writes all the Sailor data to the screen).

Next derive a new class from `Sailor`. The `Sailor_At_Sea` class has the name of the ship to which the `Sailor` has been assigned, as well as the name of the ship’s

home port. The Sailor_At_Sea class also has two constructor functions: one that does nothing and the other that receives *all* the possible Sailor data. This constructor passes the Sailor data to the base class constructor. There are also *WriteInfo* and *AskInfo* functions that call the associated *Sailor* functions before handling the derived class specific data.

The *main* function has several steps. Initially it should declare one Sailor and one Sailor_At_Sea object. Simply make up all the data and pass it into the objects when they are created. Call the *WriteInfo* functions and verify that the objects were created with the data you passed into them. Next, change the information in both objects by calling the *AskInfo* functions, and then call the *WriteInfo* function to verify that the object data has been changed.

22. Use the Mortgage Calculator class that we wrote in Chapter 7, Problem 31 (page 452) as a starting point to build your Deluxe Mortgage Calculator class. (You didn't think we wouldn't write an inherited Mortgage Calculator, did you?) In your MortCalc class, change the private data to protected. This is the only code change you will make to your MortCalc.h and .cpp files. Now use this class to derive the *DeLuxeMortCalc*. This custom calculator contains data for the taxes and insurance costs that are normally added to a mortgage payment. The new calculator has *Set* functions for the yearly homeowners insurance cost and yearly property taxes. The mortgage payment now includes the taxes and insurance costs. These two additional costs are spread evenly throughout the year. There is a new version of the *GetFormattedString* and *GetMonthlyPayment* function because they must now include the additional costs. These two functions should call the parent's functions and then perform the additional work. The basics of the loan have not changes, nor will their associated functions.

In main, create a *DeLuxeMortCalc* object, and set up a menu similar to the one described in Chapter 7's problem. Include questions for the taxes and insurance costs. Once again, there are no *cout* or *cin* statements in this new class.

23. Write a C++ program that sets up a base class BankAcct containing the protected data of a basic bank savings account (name, account number, and savings account balance). Your constructors should be set up so that the initial data is passed into them. The BankAcct class has three virtual functions: Deposit (add money to the savings account balance), Withdraw (subtract money from the saving account balance), and ShowAll (report all information).

Derive an Silver_BankAcct class from the BankAcct class that contains an ATM account number and personal identification number (PIN). The Silver_BankAcct class member functions (Withdraw and Deposit) have built-in fees associated with all the basic banking functions. The *Withdraw* function subtracts \$2.50 from the balance as a bank fee. The *Deposit* function charges \$1.00 if the deposit amount is less than \$1,000. The fee is based on the balance prior to the transaction. The Silver_BankAcct class constructor is passed all the account information (name, account number, savings balance, ATM account number, and PIN).

Derive a third class, Gold_BankAcct, from the Silver_BankAcct. The Gold objects have all the ATM data, but the ATM fees are not charged if the current savings account balance is \$20,000 or more. Gold accounts earn interest on the

account if the balance is \$5,000 or more. The interest rate is a Gold member variable, and it is passed into the Gold constructor with the other data. The interest is added into the base balance before any deposit is made.

In the *main* function, create one of all three types of bank account objects. Fill in the names and account numbers values for each object when it is created. Initialize the balances as follows:

BankAcct	\$500
Silver_BankAcct	\$1,500
Gold_BankAcct	\$25,000

Make an array of BankAcct pointers and assign each of the three object addresses into this array. Set up a loop that asks the user what type of banking transaction he or she wants. For whatever type of transaction, run a *for* loop and call the appropriate function. After each transaction, call the *ShowAll* functions, which report all data associated with each object. For example, if the user wishes to deposit \$100, then \$100 is deposited into each account. (Try to conserve screen space so that all the data may be written on the screen for each transaction.) Be sure to place error messages if the user attempts to perform an illegal transaction (i.e., not enough money in the account to make the requested withdrawal). Your program should ask the user if he or she wishes to perform another transaction. Quit the program when the user is finished.

24. Review the ParkingGarageAttendant class we wrote for the C++ International Airport in Chapter 7, Problem 30 page 452. Now we'll use this as a base class and derive a new class named PGA_ReceiptWriter that is designed to write a receipt file for the parker. In the PGA class, change the private data to protected (no other changes are allowed in the base class described and written in Chapter 7.) This new class has a default and an overloaded constructor functions (as in the Chapter 7 problem) and it has a new version of the *CalculateFee* function. This new function calls the parent's *CalculateFee* (which accesses the system time, validates the time and sets the valid flag). Then in the new *CalculateFee* function, it asks the user if he needs a receipt for his parking costs. If the user does need a receipt, *CalculateFee* calls *WriteReceipt*, which uses the time out as the name for the output file. If the time out is 10:15, the filename is 10_15.txt. The new receipt writer class must have string variables for the name of the parking facility and a nice phrase to be placed at the bottom of the receipt. These string are passed into the class via *Set* functions. If the programmer forgets to set these strings, be sure the constructor initialized the strings for a generic parking garage. The receipt includes the date, time in, time out, total time parked, and fee paid. Here is an example of a receipt. The file's name is 10_15.txt.

C++ International Airport Parking Facility
Date: October 27, 2006
Time in: 9:50
Time out: 10:15

```
Time Parked:      0:25
Fee Paid $2.00
Thanks for parking with us! See you again soon. Please stay longer
next time!
```

As in the Chapter 7 program, *main* should declare one *PGA_ReceiptWriter* object. Set the string values and then begin a parking loop. Be sure your new class is producing accurate receipts that look professional as well.

25. The C++ CoinChanger class that we wrote in Problem 35 in Chapter 7 (page 456) was a simple class that (in theory) dispenses coins given the amount of requested change. Please re-read the problem statement to refresh your memory. In this problem, we'll write a *CoinBanker* that is derived from the CoinChanger class. The *CoinBanker* will handle all your coinage needs for a vending machine. Obtain your CoinChanger.h and cpp files and change the private members to protected, so they are inherited in our new class.

The *CoinBanker* inherits the protected and public members of its parent class, CoinChanger. In other words, it inherits the coin number and types, and the functionality to dispense the correct change. Our *CoinBanker* will have the additional functionality to accept coins and keep track of the amount of money a user enters. In the *CoinBanker* class there are two constructors, as in the *CoinChanger*. The default constructor uses the parent's default setting the number of coins in the bank. The overloaded constructor is passed the number of each coin.

When you go to the US Post Office and buy stamps from the vending machine, the amount of money you've entered is shown to you on the screen. As you enter each coin, the amount is updated. If you purchase something, the cost is decremented from the total and your change is returned. Our *CoinBanker* will model this activity. It has inherited the coin counters and will keep track of the user's entered money. Write an *AcceptCoins* function that presents the user an easy way to enter coins and dollar bills. Here's one way to do this. The user inserts coins/bills by choosing from a list presented in the *AcceptCoins* function. The "Amount" value is updated with each coin. The menu choices might look something like this:

```
Enter a coin      Amount $ 0.00
(n)ickle
(d)ime
(q)uarter
(1)1 dollar bill
(5)5 dollar bill
(t)en dollar bill
($)twenty dollar bill
(x)o quit giving the banker money
```

The *CoinBanker* has a *GetCurrentAmount* that returns the amount of money the user has entered. It also has a *Reset* function that sets the current amount

to zero. The *Reset* is called typically after a transaction has been completed. Recall that the *CoinChanger* writes a log file that shows your name, a program title and gives total number of coins and total amount of money. Your CoinBanker will need to expand the information written into this log file to include the incoming money, and current total as well. You do not need to log each coin entry! Just update the coin counts and totals after the user has quit giving coins in the *AcceptCoins* function.

The *main* function will be a test arena for our CoinBanker class. The *main* function has a *do while* loop with a menu showing these banking choices and includes a quit option. The *main* function does NOT ask if the user wants to go again. The user's choice must be checked to see if he wishes to quit. These options are written from *main*—NOT from a function within the CoinBanker.

We will use text command in main for the CoinBanker. The exact commands are shown here within ()

```
MAIN MENU
=====
a (accept)           Give the coin banker coins
c (change)          Receive change from the banker
s (show current)    Show current amount of money
r (reset session)   Resets the current session to zero (get
                    ready for next customer)
q (quit)            Quit the program
```

One last detail for your consideration: the *MakeChange* function inherited from *CoinChanger* returns a *bool* indicating of it could make change for the requested amount of money. For our CoinBanker, you need to override this (i.e., make another *MakeChange* function) and it must check that the current money is the same or greater than the amount of change requested. Your *MakeChange* function follow these steps:

```
bool CoinBanker::MakeChange(float chg)
{
    //check current amount is >= requested change
    //return false (can't make change) if <
    // now call the parent's MakeChange
    bool b = CoinChanger::MakeChange(chg);
    return b;
}
```

26. The C++ CoinBanker class that we wrote in the previous problem provides a wonderful tool to incorporate into a vending machine. For this problem, let's write a VendingMachine class and sells a variety of items. In this program, you will create a VendingMachine class, which uses the CoinBanker class that you wrote in Program 25. The vending machine shows a list of products and their prices, and allows the user to purchase items, one at a time.

You should have a data file named VendingItems.txt that contains the five (5) items that your vending machine will sell. The first line of the data file contains

the name of the vending machine/item type and the following lines should have the item, and the price written like this:

```
C++ Programming Today 2nd Edition Debugging Vending Machine!
Answer to Runtime Crash
1.25
Answer to Compiler Error
0.75
Answer to Linker Error
0.50
Answer to Class Design
1.75
Program Clarification
0.10
```

Your vending machine class should have a function that opens and reads the data file. The product items are read into arrays or vectors for the item and price. (See below) The vending machine should have a *CoinBanker* object that handles the money. The vending machine should write a log file that shows the date/time it was turned on, as well as each item sold. It should also show the date/time that the vending machine was closed (i.e., before program termination).

Here is the start of how your *VendingMachine* class should be built. It should include an array of *Items*.

```
#include "CoinBanker.h"
#include <string>
using namespace std;
class VendingMachine
{
private:
    string product[5];
    float price[5];
    string vmName;           //name of vending machine
    CoinBanker banker;
    //other data variables and private methods?????
    void ReadDataFile();      //reads the data file, fills product
                               //and name
    void ShowItemsAndPrice(); //called from BuySomething,
                               //writes the items to the screen
public:
    VendingMachine();
    void BuySomething();
    void TurnOffMachine();   //passes the summary information to
                           //the FileWriter
};
```

Your *main* function will be very simple. Here is a brief outline:

```
#include "VendingMachine.h"
int main()
{
    Header();
    VendingMachine acme;
    //begin the do-while loop for making purchases
    do
    {
        acme.BuySomething();

        //ask/obtain answer "do you wish to purchase another
        //item? yes/no"
    }while( user wants to buy something)
    acme.TurnOffMachine();
    //say goodbye;
}
```

As the user buys things from the Vending Machine, the coin banker handles all of the money (accept coins, gives change, etc.). Also, the vending machine writes its own log file. (You will have two log files in this program, the Bank Log written by the CoinBanker and the Vending Machine Log, written by the vending machine.) The vending log file describes what items are purchases and the price. The system time should be obtained and written when each item is purchased. When the user is finished buying things (before program termination) the vending machine is “turned off” and summary information is written to the log file, including total items purchased and total money received. Note: You may assume there are an infinite number of items in the vending machine.

The VendingMachine’s constructor function should do the following:

1. The user should be asked for the name of the vending machine log file.
2. The constructor calls the *ReadFile* and fills the product/cost arrays. Use *#define* to hardcode the name of your products file. This should be located at the top of your VendingMachine.cpp file. If there is a problem with the file, report the error and exit.

The VendingMachine’s *BuySomething* function should perform these tasks.

1. Call *ShowMenuItem*s() to present the user with the items and their prices. (See Input file notes below!)
2. Ask which item (if any) the user wishes to purchase.(See input notes below)
3. Once the user has selected an item, call the CoinBanker’s *AcceptCoins* function.
4. The *BuySomething* must determine if the correct amount of money has been entered. It makes the calls to the CoinBanker functions, telling it to

accept coins. It then asks how much money it received by calling *GetCurrentAmount*. If the user hasn't entered enough money, report this and allow him to continue entering money or (if he doesn't want the item) have the banker return all of the money. Remember to reset when the transaction is completed. The *BuySomething* can handle only one requested item at a time. Once one transaction is completed, return to the *main* function. DO NOT include a "do another" loop here. The "do another" loop in the main function.

Here is a sample of the vending machine output log file.

```
C++ Programming Today 2nd Edition Debugging Vending Machine
Log File for 11/14/2007 at 10:45.25
Transaction 1
10:46.56 Answer to Runtime Crash      $1.25
Transaction 2
10:48.22 Answer to Compiler Error    $0.75
Transaction 3
10:49.52 Answer to Compiler Error    $0.75
10:50.46 Vending machine turned off.
Total items purchased 3
Total money received      $2.75
```

Remember that your log files should match in the money department. The vending machine log tells what was purchased and total money for each item. The CoinBanker log should report details on the money end of the operation, showing coins in/out, and total received for each transaction.

Getting Started with Visual C++ 2005 Express Edition



Appendix

A

C++ Development Environment Tools

The C++ programmer must learn and become efficient with the tools of his development environment, just as the home construction worker must know how to use his tools. Software development environments, including Microsoft's Visual C++ 2005 Express Edition, provide an integrated set of tools for writing, editing, debugging, and running programs. Table A-1 summarizes the development environment features.

■ TABLE A-1

Software Development Environment Tools

Tool	Use
Editor	The programmer writes the source code using the editor environment and saves the source file to disk. Editing tools specifically for writing C++ code are often available.
Compiler	The source code is read by a compiler, which checks to see that the source code is grammatically correct and does not violate any rules of syntax. If there are no errors, the compiler produces object or machine code. The compiler results are listed to the screen. Compiler errors in the source code are explained in a different window and a line number is given for reference. The Help section can be accessed for additional description of the error.
Linker	The linker hooks the object code and library code together and builds an executable file. This executable file runs the program.
Debugger	Once the program is running (that is, there are no compile or link errors), the debugger allows the programmer to single-step through the program, providing the ability to watch variables and to follow the flow of the program. Breakpoints can be set in the program so that the program runs and stops at these points.
Help	The package provides help reference for the C++ language as well as the error messages generated by the code.

Installing Visual C++ 2005 Express Edition

The Microsoft Visual C++ 2005 Express Edition CD that accompanies this book is used to install the Visual C++ 2005 software on your personal computer. System requirements for this software are listed below:

- Personal computer with a Pentium III-class processor, 600 MHz
- Microsoft Windows 2000 (with Service Pack 4), XP (with Service Pack 2), or Windows 2003 (with Service Pack 1)
- Minimum memory: 192 MB recommended
- Hard-disk space required:

Typical installation: 75 MB on system drive, 330 MB on installation drive
Microsoft Internet Explorer 4.01 Service Pack 1 (included)

- CD-ROM drive
- Video: Recommended 1024 × 768 High Color 16-bit
- Microsoft Mouse or compatible pointing device

It is recommended that you do the “typical” installation for both the Visual C++ 2005 Express and the Microsoft Developer’s Network (MSDN) software. The MSDN includes extensive help and reference sections.

Program Project Construction Steps

When you build programs in Visual C++ 2005, you will be working in a project and building a solution. The programmer takes several steps to set up the project/solution, and for our programs the source code file(s) are located inside the project folder. It is very important that you learn how to set up your project/solution. We’ll take you through this process one step at a time.

Visual C++ 2005 Express Projects and Solutions The majority of C++ programs consists of many source files, not just one. A programmer can organize his software into separate files according to what the program must do. Visual C++ 2005 does a nice job of keeping your files organized. We’ll learn how to work in separate files beginning in Chapter 4 and Appendix H. That said, realize that most development environments expect programs to be constructed with many source files. These environments, including Microsoft Visual C++ 2005 Express, require the programmer to create a project and to add or create the source files into the project. The project folder contains the source files (or knows where they are located). This IDE keeps track of the entire set of program files. The compile, link, and run steps are also performed via this project environment.

We show a simple example below and try to answer all your “getting started” questions. Once you work through several of the practice programs and programs in the chapters, you can return to the Visual C++ Help and review all this information on projects.

An Example: Hello World from Scratch Microsoft Visual C++ 2005 requires that the program be contained in a project. A project must be set up before a program source code can be typed in, compiled, and run. The project folder contains the source code files (including the *.cpp and *.h files) as well as several project specific files.

Warnings

1. If you are working on a jump/flash/thumb drive, you might consider working locally on the computer's hard drive to avoid waiting while your code compiles across this "slow" data pipe.
2. If you are working at home and at school, you can transport your project folders on your jump drive. Be careful not to overwrite your project as you transfer from one disk to another.
3. Be sure that you have closed your workspace/project/solution before you copy your project to your jump drive.



In this example, we set up a new project in Microsoft Visual C++ 2005 project. The program writes "Hello World!" to the output screen. The project has the following characteristics:

- Designated Working Directory: C:\C++PT
- Project name: JohnstonDemo1
- Program file: hello.cpp

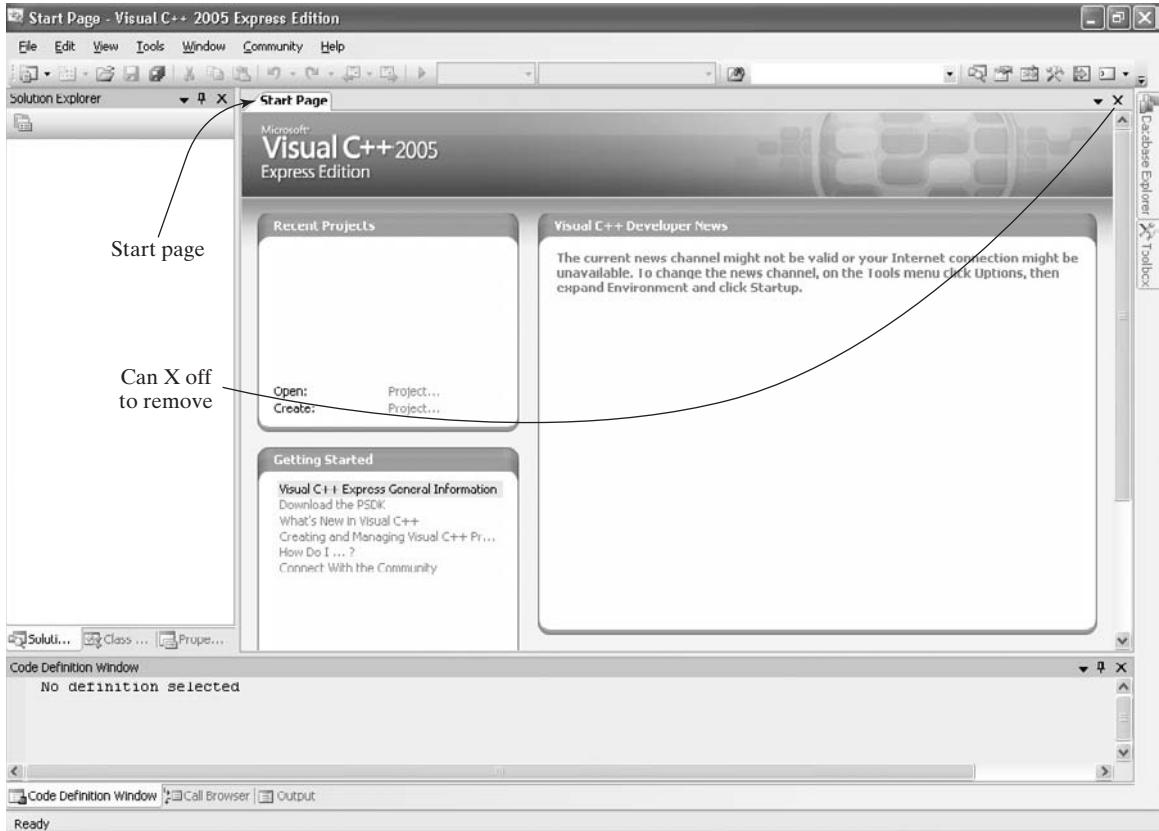
Start the Visual C++ 2005 Program Visual C++ 2005 may be accessed via the Start menu. You may also add it to a toolbar or create a shortcut to the program and place the shortcut on your desktop. Make use of the **Start** menu: **Start** → **Programs** → **Microsoft Visual C++ 2005 Express Edition** → **Microsoft Visual C++ 2005 Express Edition**. You will see the program and the first time you open it, you'll see the Start Page. See Figure A-1

Step 1: Create the project. You need to do this step *only once* for each program! Once you have it set, you can work on your program as often as necessary.

Under File → **New** → **Project** → You'll see the New Project window. See Figure A-2.

In this window select **Win32 Project type** in the left pane and **Win32 Console Application** in the right pane. Browse (see NOTE!) to your desired directory (here we selected the C++PT folder) and type in the name of your project (here it is JohnstonDemo1) in the top text field. Watch as you type in the project name; it is echoed in the Solution Name field. This creates a project folder by this name. Hit the **OK** button.

NOTE! Visual C++ has a default location for your project—but most programmers wish to place their projects in a different folder, such as a course name folder. Pay attention to this location! It is a common problem that beginning students can't find their projects because they didn't note the location!

**Figure A-1**

Microsoft Visual C++ 2005 Express Edition start page.

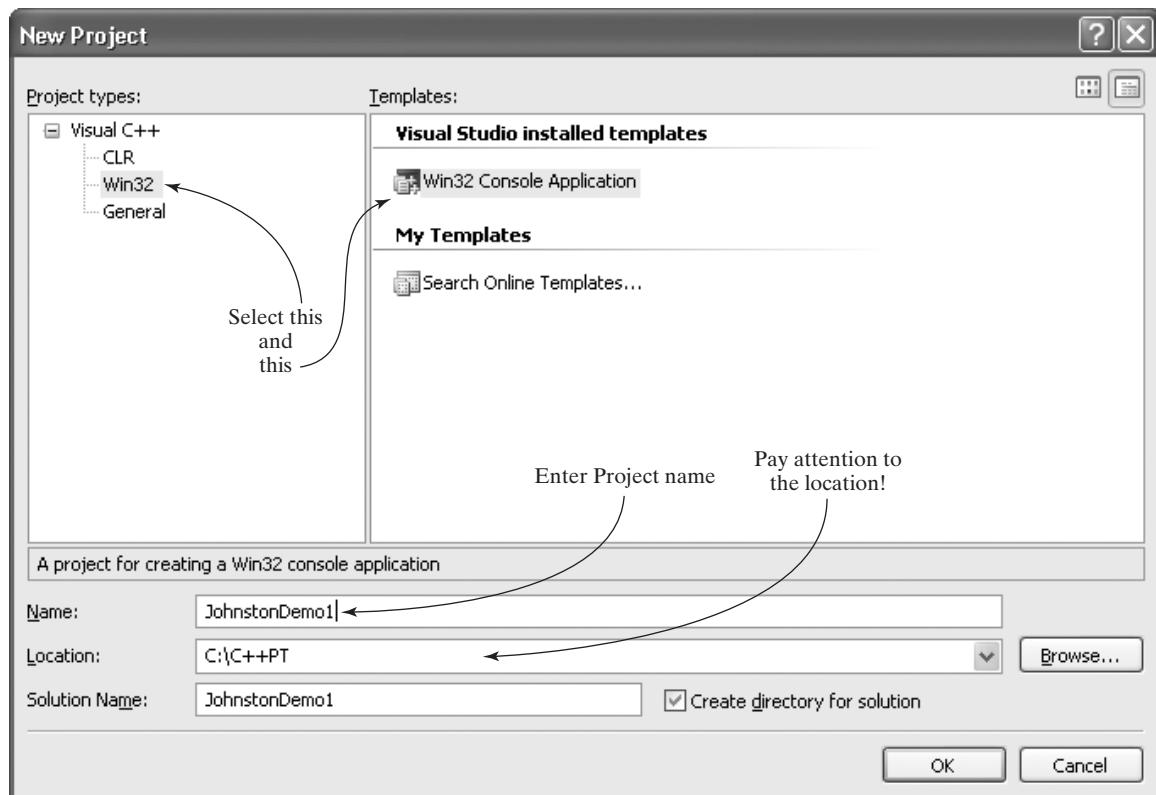
Now you'll see the **Win32 Application Wizard** for JohnstonDemo1. See Figure A-3. Select the **next** button.

The **Application Settings** window is shown next (See Figure A-4.) VERY IMPORTANT!!! Select a Console application and an **Empty Project** check box. Hit the **Finish** button. We need an empty project because we will build our source code from scratch. (Visual C++ can build projects of all sorts that has pre-written source code containing headers and other code. We are not going to use this C++ format.)

When you finish these steps, you should see your Visual C++ 2005 Solution. Note that there are Header, Resource and Source Files in the window on the left.

Step 2: Create a new source file and type in your code. There are several ways for us to create a source file for our project. We'll use an easy way. Right click on the **Source Files** folder in the **Solution** window on the left. You'll see a popup menu—select the **Add : New Item**. Select the **Code** category, and **C++ File (.cpp)** template. Enter the name of the source file. Press the **Add** button. Here we name our file **Hello.cpp** (See Figure A-5.)

Now enter the following source code. Your window should look like Figure A-6 on page 535.

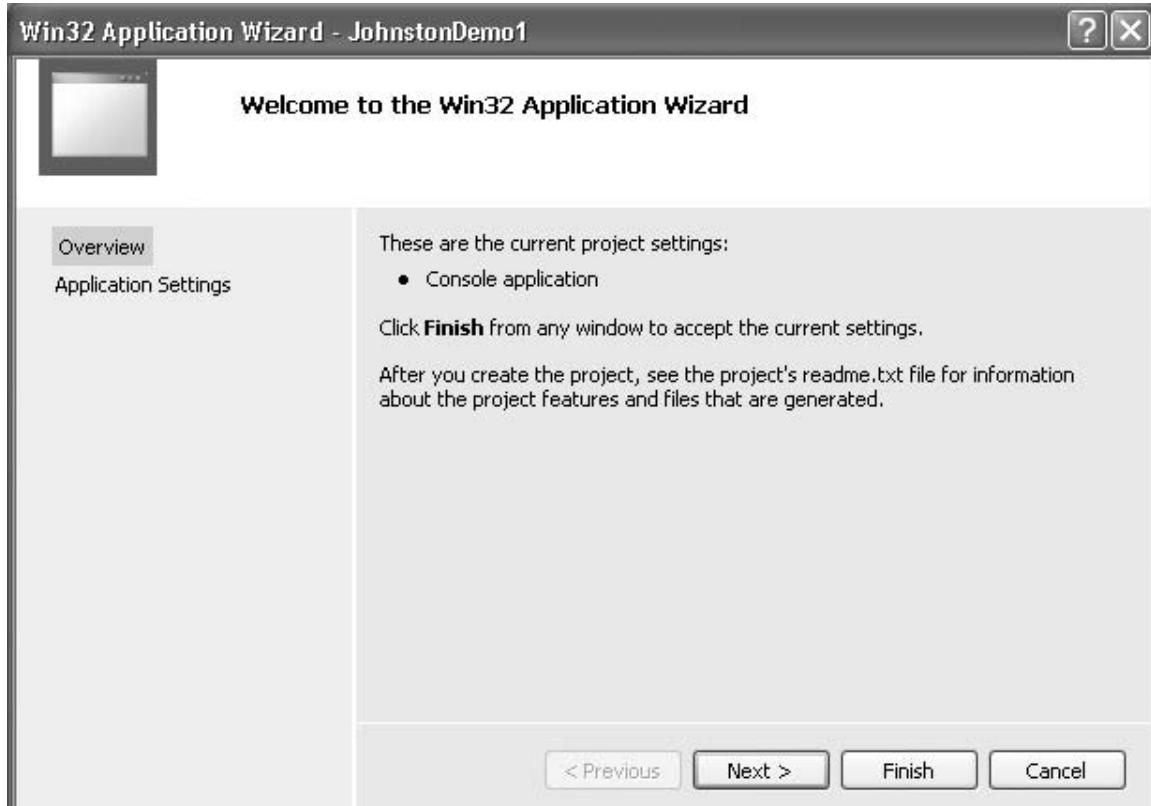
**Figure A-2**

The New Project window.

```
//Getting started with Hello World.
//This is our first program.
#include <iostream>
using namespace std;
int main()
{
    cout << "\n Hello World!      \n" ;
    return 0;
}
```

Notice that once you start typing in the window, the file name at the top of the window changes to Hello.cpp*. The * indicate that this file has new contents since it was last saved.

Step 3: Compile your source code. To compile the source code, you should hit the **Build : Build Solution** menu item. Visual C++ 2005 reports the build progress in the lower window. (See Figure A-7.) As you can see we had Build: 1 succeeded and 0 failed, so we are good to go! Any compile errors are shown in this lower window. You may double click the line in the error window, and it will put

**Figure A-3**

The Win32 Application Wizard.

the cursor on the line in the source code where it believes the problem exists. If there are no errors, it will show the message seen in Figure A-7 (page 536).

Note on compile errors: if the line where the error is being reported looks good, examine the line above it. Sometimes the problem is on the previous line!

Step 4: Run your program. To execute the program, you should select the **Debug : Start Without Debugging**.

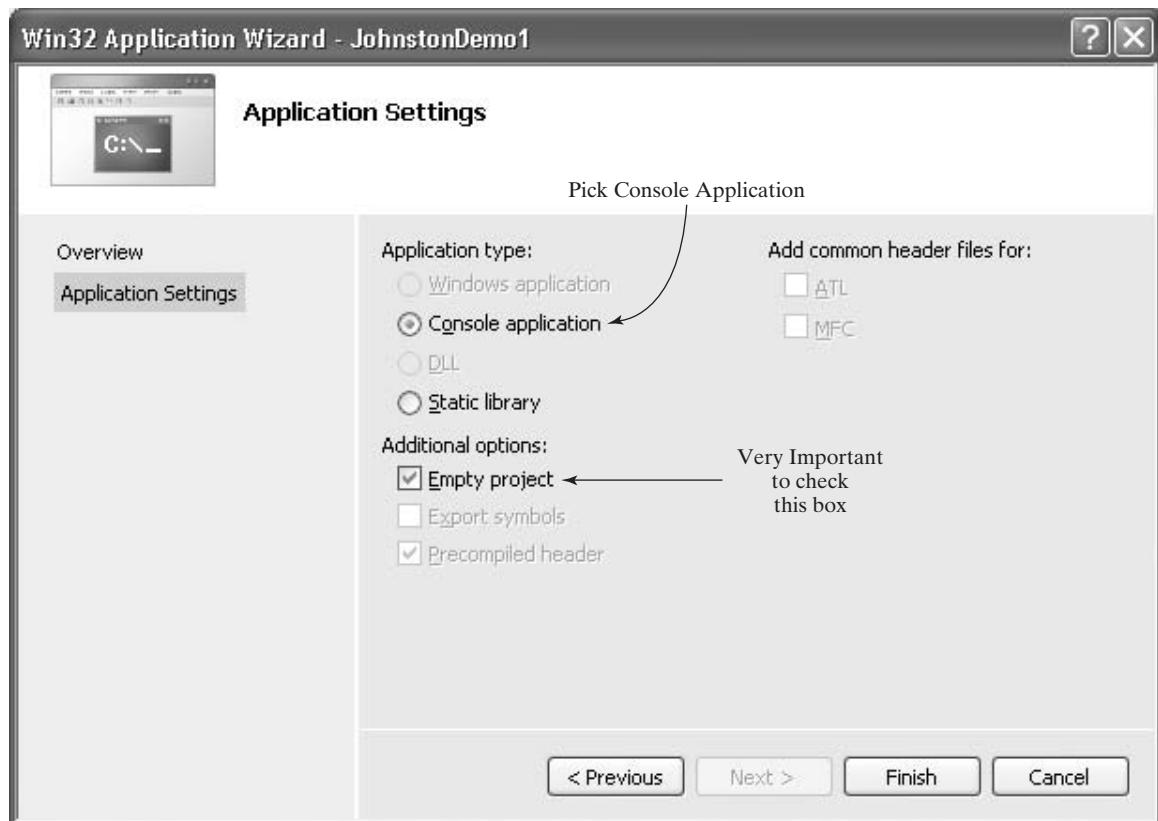
The console window will be displayed by Visual C with your program results (see Figure A-8). When your program is finished, it will display the following message:

Press any key to continue ...

When you hit the Enter key, this window disappears.

Step 5: All finished for the day. To exit Visual C++ 2005, you should **File→Save All**, and close your solution.

Step 6: Check to see what files Visual C has set up for you. Because we are just beginning with Visual C++, now is a perfect time to examine the files that Visual C++ has set up in the project folder. Using My Computer or Explorer examine the files in your JohnstonDemo1 project (See Figure A-9, page 537.) You will see Debug folders, as well as the Hello.cpp file. There are several other files that have

**Figure A-4**

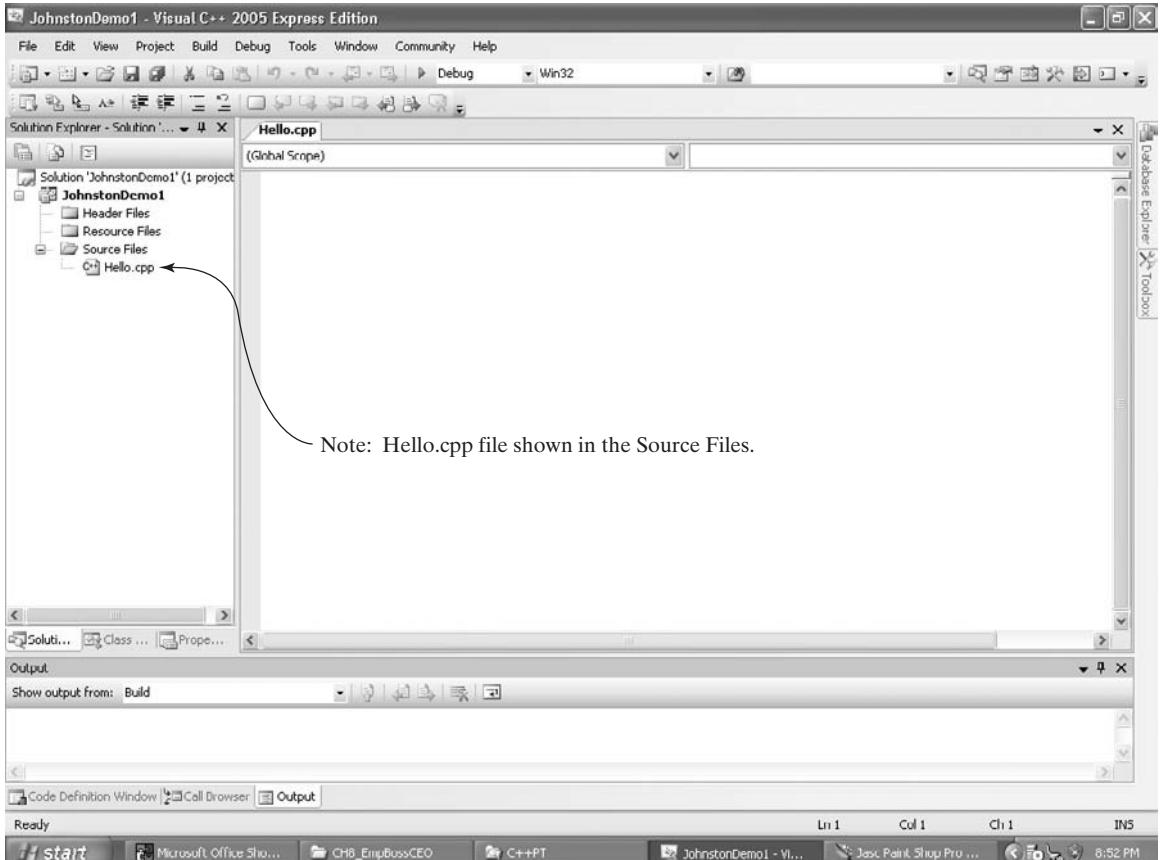
The Application Settings window; be sure to select an Empty Project!

been created for your project. We won't worry about these now. Just realize that they exist, and allow them to stay in these folders.

Visual C creates a project and the default build mode is Debug. When compiling a program in the debug configuration, Visual C provides the necessary "hooks" for the debugger. This provision makes the program executable file a fairly large one. (Note: once your program is working well, you may select the configuration option of Release, which compiles and builds your program without the debugging hooks. At this point, the project folder should contain a Release folder that contains an *.obj and an *.exe file. The size of the executable file is smaller here.)

Step 7: Save early, save often. You need to copy your project folder to other locations for safekeeping. The debug folders can be deleted with no problem if you need to save space. They can always be rebuilt if necessary.

Exit Warning It is a *bad idea* to close Visual C++ by pressing the X button in the top right corner of the window. Save your workspace and then close it. By saving and closing your workspace, you ensure that Visual C++ is shut down properly, your files and workspace are saved, and everything can be located when Visual C++ is restarted.

**Figure A-5**

We have created a new source file named Hello.cpp. Notice it is shown in the Source folder on the left.

Step 8: Start working on your program again.

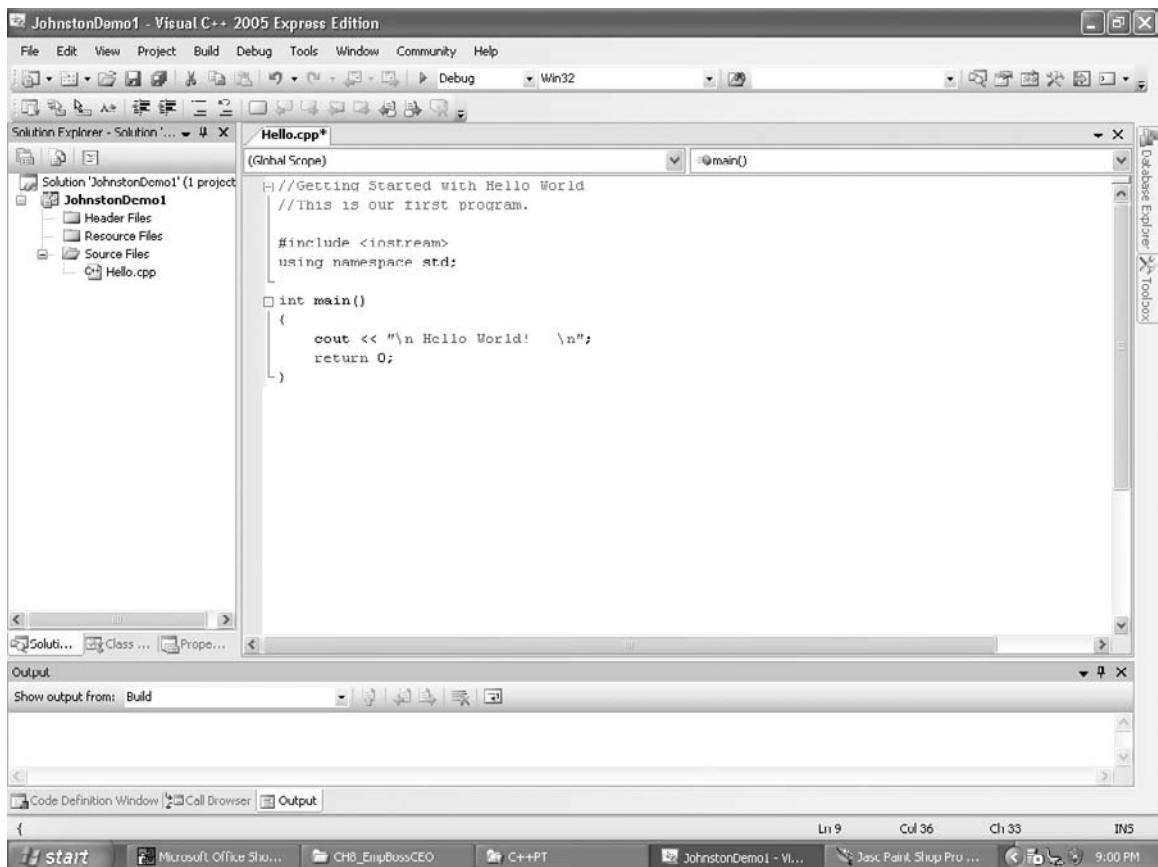


*This part of Appendix A is crucial. When you start working on your project, you must open your project/solution, **not** your cpp file! Do NOT double click your cpp file to begin working on a program!*

There are two ways to get started again once you have a project built.

First way: Open Visual C++ 2005, **File**→**Open**→**Solution/Project**→**Browse** to your project folder and select the *.sln file (solution file). (See Figure A-10.)

Second way: Using the My Computer or Explore feature of your operating system, browse through the directory structure and locate your project file with the .sln extension. Double click this and you'll bring up Visual C++ 2005 with your corresponding project. This step brings you right into your project and you are ready to go.

**Figure A-6**

Enter these lines of code into Hello.cpp.

An Important Warning

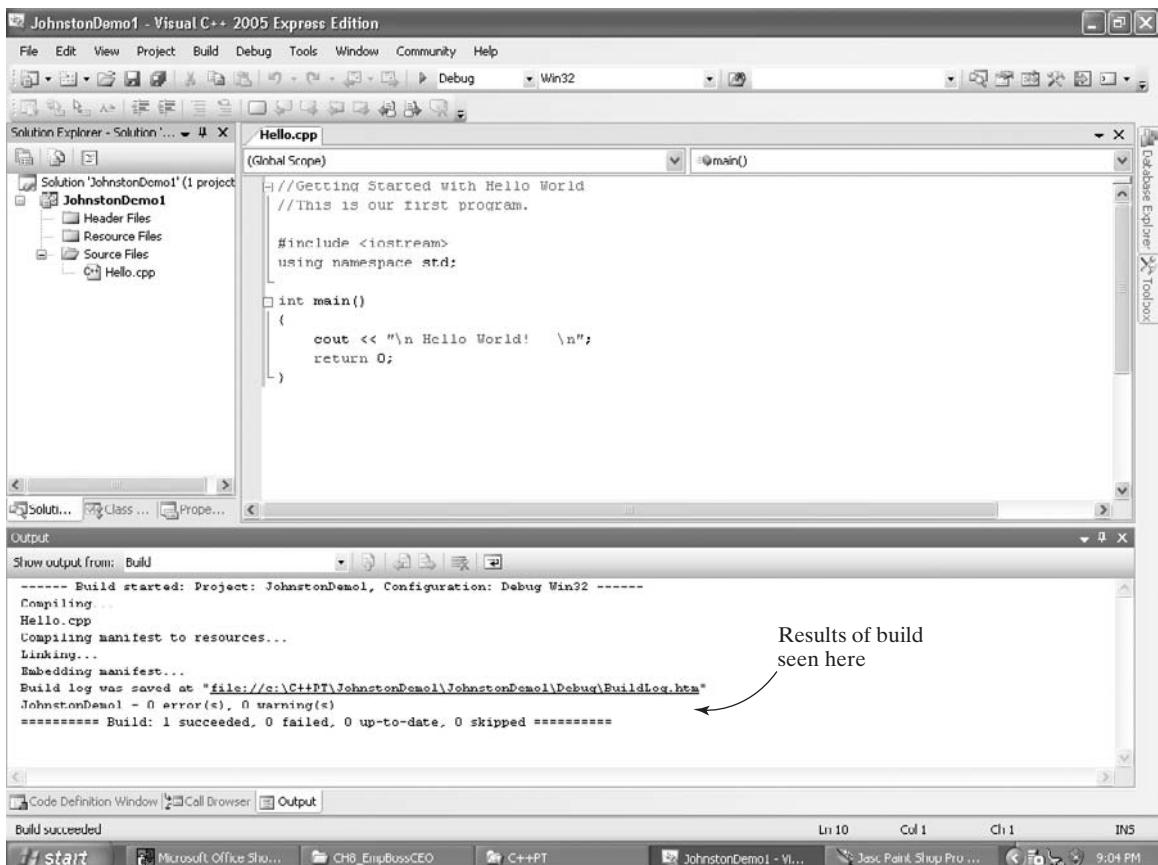
Opening the source code file (with a double click while in My Computer or Explorer) automatically opens Visual C++ 2005. But be warned! You do not have an active solution! This is BAD! (See Figure A-11, page 538.)

Notice that the window on the left shows (0 Projects). If you try to compile this file, you won't see the **Build** menu! (Look again at Figure A-11.) No **Build** menu should be a clue that you've done something wrong! Go back and open your project/solution, and you'll be ready to work on your project!

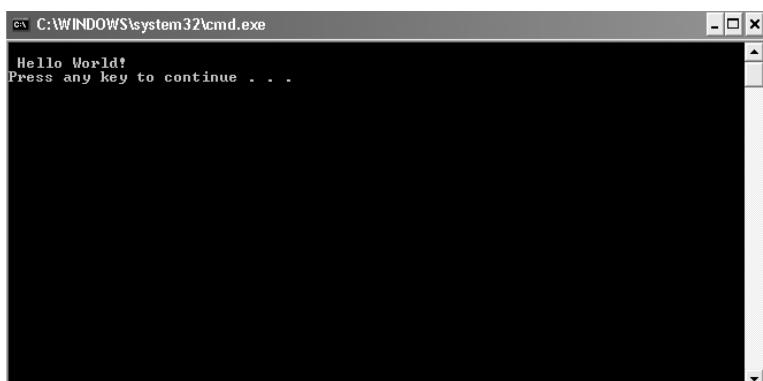


Help

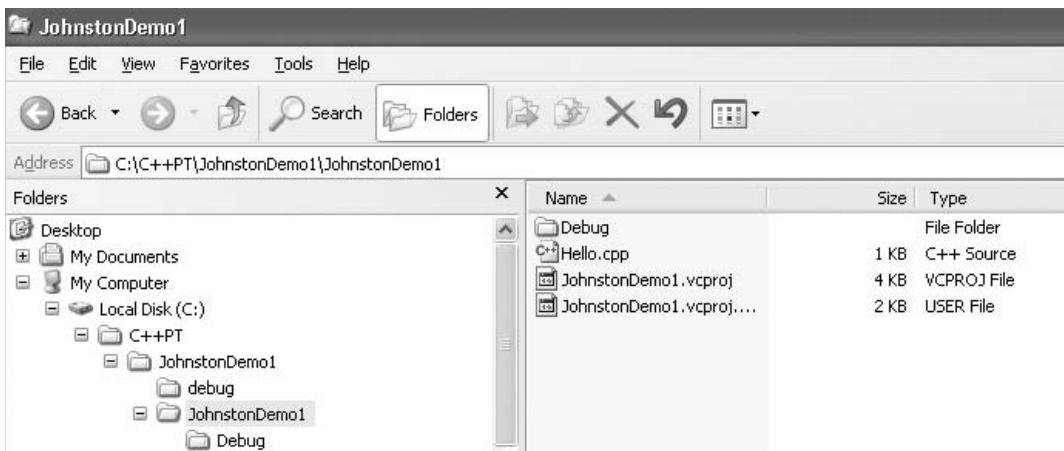
Visual C++ provides an extensive Help system that may be accessed in several ways. The first and most obvious way is through the pull-down menu on the Visual C++ main window. If you follow the links to the Search section of the help, you'll

**Figure A-7**

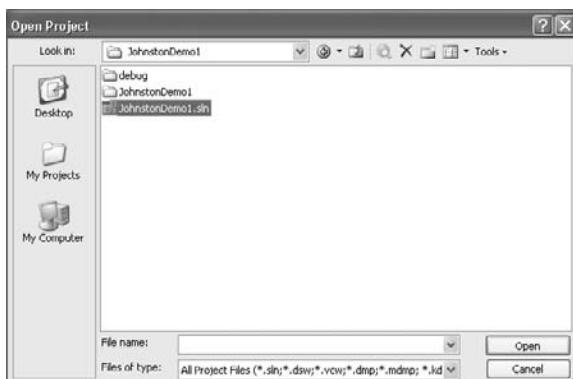
Visual C++ 2005 reports the build progress in the lower window.

**Figure A-8**

Program results are shown in the console window.

**Figure A-9**

Folder structure for the JohnstonDemo1 project created by Visual C++ 2005.

**Figure A-10**

The Open Project dialog box is presented when you do **File** → **Open** → **Solution/Project**. Select the Solution file (*.sln).

see an extensive reference section. Figure A-12 shows you that there are references for Visual C++ as well as for the language itself.

The Index tab provides a quick and easy way to find a topic if you know what you are looking for. For example, if we look for information on the vector class, there is a large amount of information including sample code for class members (See Figure A-13.)

The Search tab allows you to enter a phrase that will search all the reference material for you and provide a list for you to examine. For example, you can enter “formatted output” or “memory leak,” and the Search engine produces a multitude of articles for you.

Finally, whenever you have either a compile or link error, a number is always associated with the error. If you highlight the error number and press F1, the documentation on the error is presented for you. You should spend some time looking

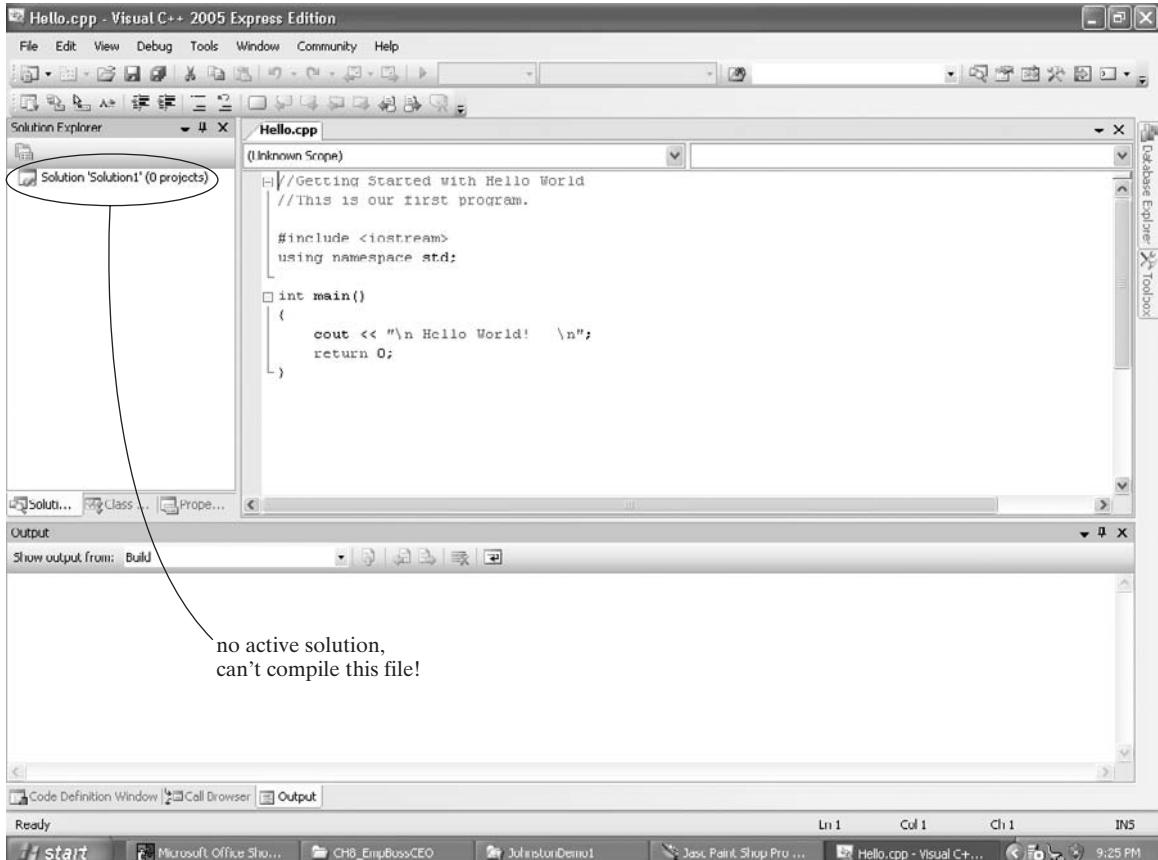


Figure A-11

Clicking on the *.cpp file opens Visual C++ 2005, but does not give you an active solution! You can not compile and build this file!

through the different Help options. The Visual C++ Help is a powerful feature of this software. Learn how to use it!

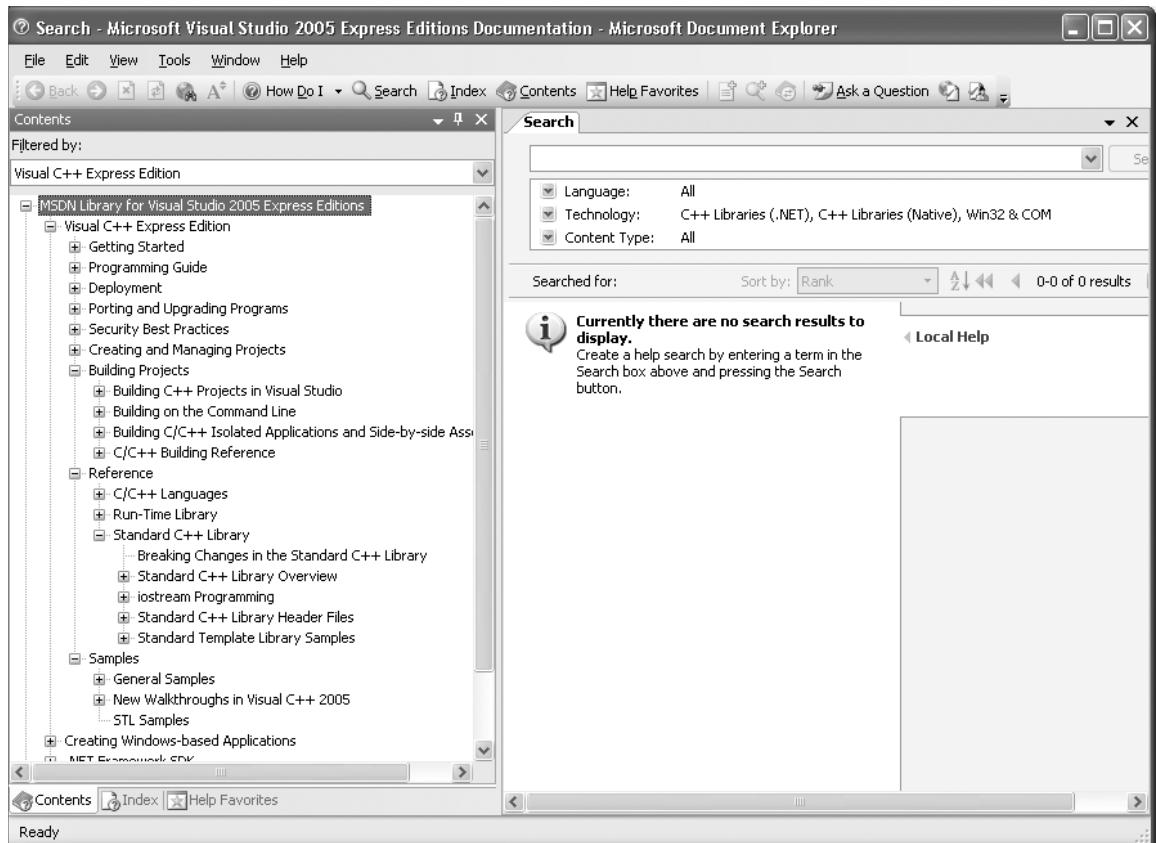
Practice Building Projects!

1. Create a project called QUOTES and a .cpp file called FourScore. Here is the program for you to enter in your FourScore.cpp file:

```
//Gettysburg Address.

#include <iostream>

using namespace std;
```

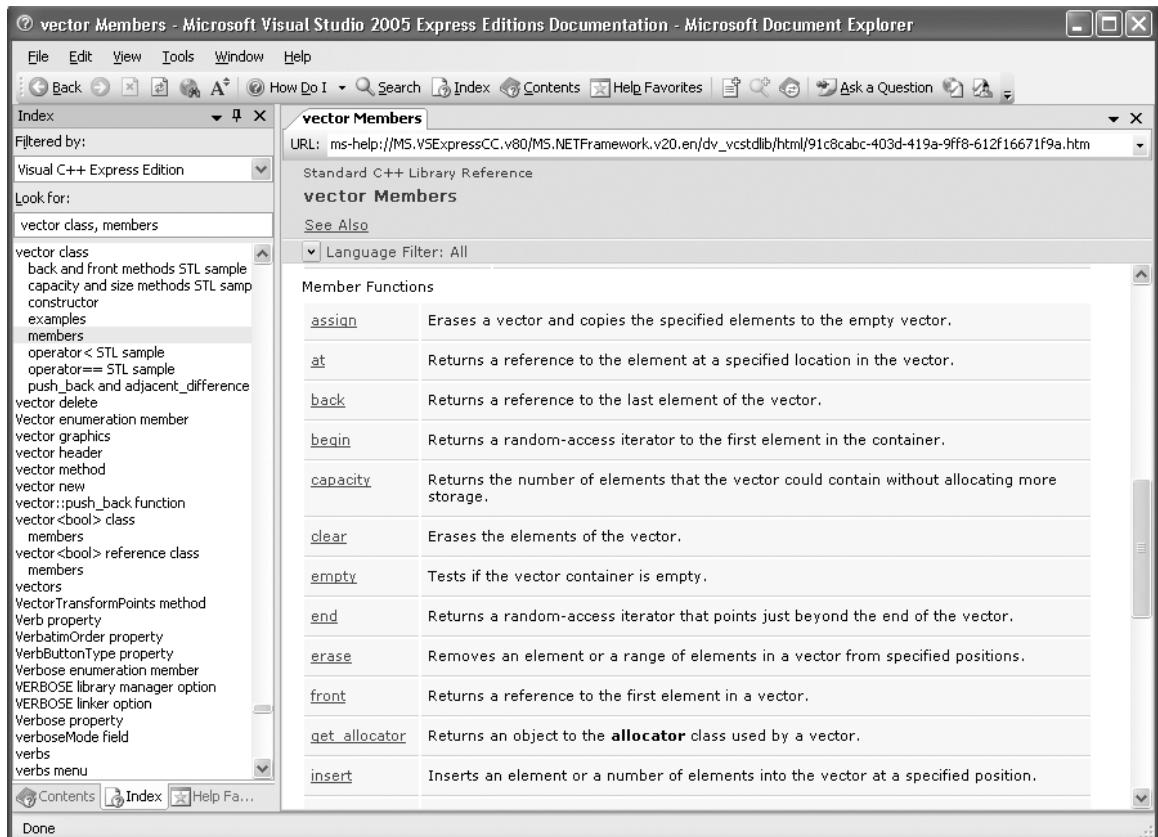
**Figure A-12**

There is an extensive Visual C++ reference as well as Standard C++ reference located in the Help section.

```
int main()
{
    cout << "\n Four score and seven years ago our fathers "
        << "\n brought forth on this continent, a new nation,"
        << "\n conceived in liberty and dedicated to the proposition"
        << "\n that all men are created equal. \n\n";
    cout << "\n\n Author: A. Lincoln \n";
    return 0;
}
```

2. Create a project called BUGS and a .cpp file called APoem. Enter the following poem in the APoem.cpp file:

```
//BUGS Program.
#include <iostream>
```

**Figure A-13**

The vector class is fully referenced in the Visual C++ 2005 Help section.

```
using namespace std;
int main()
{
    cout << "\n\n Spiders, Roaches, Ants and Ticks"
        << "\n Centipedes and Flies"
        << "\n Have many legs to walk upon."
        << "\n Just so they walk outside.";
    cout << "\n\n Author: B. Johnston \n";
    return 0;
}
```

3. Go back to the QUOTES project and add a title line that will be shown in the output window. The title should be something about the Gettysburg Address.
4. Return to the BUGS project and add your own verse.

C++ Keyword Dictionary

Appendix B

A total of sixty-three keywords are part of the current ISO/ANSI C++ Standard. The ISO C Standard has thirty-two keywords, and the C++ Standard added thirty-one. The language reserves these keywords, and they cannot be used as program identifiers—that is, keywords cannot be used in a program as function or variable names. The sixty-three C and C++ keywords are listed below.

asm	auto	bool	break	case	catch
char	class	const	const_cast	continue	default
delete	do	double	dynamic_cast	else	enum
explicit	export	extern	false	float	for
friend	goto	if	inline	int	long
mutable	namespace	new	operator	private	protected
public	register	reinterpret_cast	return	short	signed
sizeof	static	static_cast	struct	switch	template
this	throw	true	try	typedef	typeid
typename	union	unsigned	using	virtual	void
volatile	wchar_t	while			

The thirty-one keywords specific to C++:

asm	bool	catch	class	const_case	delete
dynamic_cast	explicit	export	false	friend	inline
mutable	namespace	new	operator	private	protected
public	reinterpret_cast	static_cast	template	this	throw
true	try	typeid	typename	using	virtual
					wchar_t

Each of the sixty-three keywords is presented in Table B-1 with a brief definition, and most keyword entries contain sample usage. Where applicable, chapter references are also listed. A few topics are presented only in the keyword dictionary, and the reference is shown as “C++ Ref,” which indicates that the reader should consult a complete C++ reference for discussion and examples. A brief definition of terms frequently used in the dictionary is presented at the top of page 542.

access modifier	controls how a variable may be used or changed in a program.
access specifiers	controls how data or functions are available (or not) to the program
casting operator	transforms one type of variable into another
control statement	directs flow of the program execution steps
data type declaration	statement that creates data variable(s) and associated memory for storage of variable values
jump statement	causes execution of the program to move to another part of the program
label statement	a valid identifier followed by a colon
modifier	alters or provides additional capability for a keyword
operator	a word or a symbol that is reserved in C++ and that performs a specific task
storage specifier	dictates the manner in which the compiler stores the variables

TABLE B-1

Keyword, Use, Example, and Reference

Keyword	Use and Example	Refer to Chapter
<code>asm</code>	To embed assembly language directly into C++ programs. <code>asm{</code> instruction sequence <code>}</code>	C++ Ref
<code>auto</code>	Data type declaration for local variables. <i>Note:</i> Locally declared variables are assumed to be automatic by default and auto declaration is not needed. Therefore, <code>auto</code> is one of the least used keywords. <code>auto int x;</code>	2
<code>bool</code>	Data type declaration for true or false valued variables. <code>bool OK, Flag; //declare boolean variables</code> <code>OK=true; //assign true to OK</code> <code>Flag=false; //assign false to Flag</code>	2
<code>break</code>	A jump statement commonly used in switch statements. <i>See</i> switch.	3
<code>case</code>	A label statement used in a switch statement. <i>See</i> switch	3
<code>catch</code>	Used in exception handling. Statement used to handle the error. <i>See</i> try.	C++ Ref
<code>char</code>	A data type declaration for variables that hold characters. Can be single variable or array, which is a character string. <code>char answer; // single variable</code> <code>char name[50]; // character array or C-string</code>	2, 6

class A declaration used for creating classes. It defines a new type combining data and functions. 7,8

```
class X
{
    private:
        int a,b;
    protected:
        DoSomething();
    public:
        X();
        DoSomethingElse();
};
```

const An access modifier that controls how variables may be changed. 2
A program cannot change the value of a **const** variable after it has been initialized. The **const** modifier may be used in variable declarations or in function header lines to keep the function from changing a value.

```
// variable cannot be changed
const double pi=3.14159265;
// function cannot change this
Functional (const char[]);      // char string
```

const_cast A casting operator that overrides **const** and volatile declared variables. As shown here, the **const_cast** allows the **const** pointer that contains the address of pi to be assigned into a non-const variable, n. C++ Ref

```
void ChangePi(const double *);
int main()
{
    double pi = 3.14159;
    cout <<"pi = " << pi;           //writes 3.14159
    ChangePi(&pi);
    cout <<"\npi = " << pi;       //write 27
    return 0;
}
void ChangePi(const double *newpi)
{
    double *n;
    n = const_cast<double *>(newpi);
    *n = 27;
}
```

Continued

TABLE B-1

Keyword, Use, Example, and Reference (Continued)

Keyword	Use and Example	Refer to Chapter
<i>continue</i>	A jump statement used in loops. A <i>continue</i> statement forces the next iteration of the loop and will skip any code remaining in the loop. As shown here, the <i>for</i> loop is used to count the letters in the string. If a space is found, a <i>continue</i> is used to skip the remaining statements and iterate the loop. <code>char str(40) = "A bird in the hand is worth two in the bush.";</code> <pre>int count_letters; for(i = 0; i < 40; ++i) { if(str.at(i) == ' ') continue; ++count_letters; }</pre>	3
<i>default</i>	A label statement used in a switch statement. If none of the cases in the switch statements are true, the statement(s) in the default block are performed. See <i>switch</i>	3
<i>delete</i>	An operator that frees memory allocated by the <i>new</i> operator. It cannot be used with any other type of pointer. See <i>new</i>	C++ Ref
<i>do</i>	A loop/iteration control statement used in conjunction with a <i>while</i> statement. A <i>do while</i> loop will execute the loop statements once before checking the condition. The loop statements are repeated as long as the condition is true. <pre>int number = 0; do { cout << "\n The number is " << x; ++ number; } while(number < 100);</pre>	3
<i>double</i>	A data type declaration for variables that hold numeric values with decimal precision. The ISO C++ Standard states that a double must maintain at least ten digits of decimal precision. <pre>double x, y;</pre>	2

dynamic_cast	A casting operator that performs run-time verification of a polymorphic casting operation. A <i>dynamic_cast</i> is used with base and derived class pointers. One example that is legal: it is possible to cast a derived class pointer into a base class pointer.	C++ Ref
	<pre>BaseClass *pB, Bobject; DerivedClass *pD, Dobject; pB = &Dobject; //base class pointer points to //derived object // legal to cast base into derived pointer pD = dynamic_cast < DerivedClass * > pB;</pre> <p>Many variations of this cast are legal, but here is an illegal cast:</p> <pre>BaseClass *pB, Bobject; DerivedClass *pD, Dobject; pB = &Bobject; //base class pointer points to base object pD=dynamic_cast<DerivedClass *>pB; // can't do this</pre> <p>You cannot cast a base object into a derived object.</p> <p>Consult a C++ reference for further explanation.</p>	
else	A conditional control statement used with an if statement. There are two formats using the <i>else</i> keyword. It may be used in conjunction with an if or used separately. There can be only one else condition block for an if. If none of the conditional statement in the if statements are true, the statement(s) in the <i>else</i> block are performed.	3
	See if.	
enum	A data type declaration statement that creates an integer-based (numeric) list of constants. The constant values are assigned integers beginning with 0 (zero) unless otherwise specified.	C++ Ref
	<pre>enum DogBreed { Husky, Heeler, Collie, Boxer, Mixed }; enum Cards { two = 2, three, four, five, six, seven, eight }; int main() { DogBreed MyDog; MyDog = Husky; Cards MyCard; MyCard = two; if(MyCard == three) cout << "\n The card is a three";</pre>	
explicit	The <i>explicit</i> keyword is used only with converting constructors. It allows the programmer to create nonconverting constructors. Constructors with one input argument are known as converting constructors. For example, if a class constructor can be called like this:	C++ Ref

■ TABLE B-1

Keyword, Use, Example, and Reference (Continued)

Keyword	Use and Example	Refer to Chapter
	<pre>class NewClass { private: int x; public: NewClass(int y) { x = y; } }; int main() { NewClass NC(5); // could also NewClass NC = 5; }</pre> <p>It is possible not to allow this automatic conversion by writing:</p> <pre>explicit NewClass(int y) { x=y; }</pre> <p>in the class definition.</p>	
<i>export</i>	The <i>export</i> keyword is used in a template declaration so that other files are allowed to use the declared template. When an export template is declared in another file, the second file may use the original template without duplicating the template definition.	C++ Ref
	<pre>// File1.cpp Template function definition // Place the y into the x[num] element of the array template <class A> void PutIntoTheArray(A *x, A y, int num) { x[num] = y; } // File2.cpp exported to another file export template <class A> void PutIntoTheArray (A *x, A y, int num);</pre>	
<i>extern</i>	A storage class specifier for variables to be shared by routines in multi-App. H file or multilibrary programs. A variable is originally declared globally in one file and can be globally <i>extern'd</i> in another file. For example:	
	<pre>//File1.cpp double x; // declared globally //File2.cpp extern double x; // File2.cpp can now use // the x variable.</pre>	
<i>false</i>	The keyword <i>false</i> is a Boolean constant with a zero value. See <i>bool</i> .	2
<i>float</i>	A data type declaration for numeric values with decimal precision. The C++ language requires five digits of decimal precision with a float variable.	2

<i>for</i>	A loop/iteration control statement. The <i>for</i> statement contains three parts: initialization, conditional check, and increment.	3
	<pre>cout << "\n Counting 0 to 9 "; int i; for(i = 0; i < 10; ++i) { cout << "\n The number is " <<i; }</pre>	
<i>friend</i>	A friend function is declared in a class declaration. A friend function has access to all private and protected members of that class but is not a member of the class.	C++ Ref
	<pre>class A { private: int x, y; public: friend void WriteValues(); //declared as friend } void WriteValues() // since it's a friend, // it can access x and y { cout << "\n The values are " << x << " and " << y; }</pre>	
<i>goto</i>	A jump statement that requires an associated label statement. The goto and its label must be in the same function. The goto can be used to skip statements.	C++ Ref
	<pre>int count; // code that establishes value for count if(count < 10) goto next1; // code that will be skipped if count is less than 10 next1:</pre>	
<i>if</i>	A conditional/decision control statement.	3
	<pre>if(a > 0) { // statements } else if(a == 0) { // statements } else { // statements }</pre>	

TABLE B-1

Keyword, Use, Example, and Reference (Continued)

Keyword	Use and Example	Refer to Chapter
<i>inline</i>	The <i>inline</i> keyword is placed in function declarations. This placement causes the compiler to place the function at the point in the software where the function is called.	C++ Ref
	<pre>inline void SwitchEm(float *x, float *y); //prototype int main() { float sum, a = 4.0, b = 5.0; SwitchEm(&a,&b); // call to the function } inline void SwitchEm(float *pX, float *pY) { float temp = *pX; *pX = *pY; *pY = temp; }</pre>	
<i>int</i>	A data type declaration for whole numbers. The range of values for an integer variable depends on the machine architecture.	2
	<pre>int number; // minimal range -32767 to 32768 // range on Visual C++ ~-2 billion to 2 billion</pre>	
<i>long</i>	A data type declaration for integers and doubles. In some machine architectures, the long declaration provides twice the memory space (and increases the range) for the variable values.	2
	<pre>long int number; // minimal range ~-2 billion to 2 billion long double x; // minimal range 10 digits of precision</pre>	
<i>mutable</i>	The <i>mutable</i> keyword, when used in a class declaration, allows a member of an object to modify a variable modified by <i>const</i> .	C++ Ref
<i>namespace</i>	The <i>namespace</i> defines the scope or region of a program. It is possible to have separate namespace regions to avoid confusion—especially if functions have the same name. Here is a namespace region called <i>MyOwn</i> that contains a <i>rand</i> function.	2 C++ Ref
	<pre>namespace MyOwn { int rand(); }; int MyOwn::rand() { return 42; // my number is always 42 } //call statement would be int n = MyOwn::rand();</pre>	

<i>new</i>	The <i>new</i> operator allocates memory at run-time. The <i>new</i> operator returns a pointer to the allocated memory. The <i>delete</i> operator is used to free the memory and should be used only with valid pointers. If adequate memory is not available, this operation fails and in Visual C++ returns a NULL pointer. Other C++ compilers may issue a <i>bad_alloc</i> exception. (The reader should refer to a C++ reference for complete discussion.)	C++ Ref
	<pre>class Xclass { // description }; int main() { int *ptr; // pointer to an integer Xclass *xptr; // pointer to an Xclass ptr = new int; // allocate memory for an integer xptr = new Xclass; // allocate memory for Xclass object delete ptr; // free the memory delete xptr; }</pre>	
<i>operator</i>	Creates a new definition of a function for a given operator. It is possible to create class member operator functions and non-class member operator functions—or overloaded friend functions. The <i>operator</i> keyword is used in the function prototype. This example shows how to build a class member overloaded operator function.	7 C++ Ref
	<pre>class Xclass { public: void operator ++(); // create a ++ operator for the Xclass. }; void Xclass::operator ++() { //function code here } (The reader should consult a C++ reference for a description of a non-class operator function.)</pre>	
<i>private</i>	An access specifier used in classes. The private members are accessible only by other members of the class. <i>See class.</i>	7.8

TABLE B-1

Keyword, Use, Example, and Reference (Continued)

Keyword	Use and Example	Refer to Chapter
<i>protected</i>	An access specifier used in classes and inheritance. The protected members are accessible by members of the class and are inherited in derived classes. See class.	7,8
<i>public</i>	An access specifier used in classes. The public members are accessible by all parts of the program. See class.	7,8
<i>register</i>	A storage specifier for variables that allows the program faster access to variables. Register memory is located so that the program requires fewer steps to access the stored values.	C++ Ref
<i>reinterpret_cast</i>	A casting operator that changes one data type into a completely different data type. <code>char *s = "this is a string"; int pinteger; pinteger=reinterpret_cast<int>(s);</code>	C++ Ref
<i>return</i>	A <i>return</i> is a jump statement used to return program execution from a calling function to the called function.	2,4,5
<i>short</i>	A data type modifier for integers. A short int is half the number of bytes as the normal int. In Visual C++ an int is 4 bytes. A short int is 2 bytes and has a range from -32767 to 32767.	2
<i>signed</i>	A data type modifier for integers and chars. A <i>signed</i> variable includes negative values.	2
<i>sizeof</i>	An operator that computes the actual size, in bytes, of any variable or data type. Can be used with either data type or data variable, such as: <code>double x; cout << sizeof(double); //writes 8 since doubles are 8 bytes cout << sizeof(x); // same result as above</code>	2,5
<i>static</i>	A storage class modifier for local variables so that variable values are maintained until program termination.	4
<i>static_cast</i>	A casting operator can be used for standard casting or nonpolymorphic cast of a base class into a derived class. The <i>static_cast</i> is essentially the C language casting operator. <code>int n = 5; double x; x = static_cast <double> (n); float f; f = static_cast<float>(3.3);</code>	2 C++ Ref

struct	A declaration used to create data structures, such as: <pre>struct Ball { float radius, surface area; };</pre>	C++ Ref
switch	A conditional/decision control statement. <pre>switch(variable) { case 1: // code executed if variable's value is 1 break; case 2: // code executed if variable's value is 2 break; default: //code executed if no case values are found }</pre>	3
template	A declaration statement used to create a generic class or function.	C++ Ref
this	When a member function is called in objects and classes, there is an implicit pointer. The implicit pointer is the “this” pointer. It points to the invoking object. For example: <pre>class MyClass { public: void Function1(); }; int main() { MyClass MyObject; MyObject.Function1();</pre>	C++ Ref
	When the Function1 function is called, the “this” pointer points to the MyObject object.	
throw	Found in exception handling, a <i>throw</i> statement must be executed when an error is found. See try.	C++ Ref
true	The keyword <i>true</i> is a Boolean constant with a nonzero value. See bool.	2

TABLE B-1

Keyword, Use, Example, and Reference (Continued)

Keyword	Use and Example	Refer to Chapter
<i>try</i>	<p>Keyword found in exception handling. The <i>try</i> is a block of statements used to monitor code when looking for errors. When an error is encountered, we throw an exception, which is “caught” by the <i>catch</i> statement. In this example, we ask the user to enter a positive number. We check the number in the <i>try</i> block and if it is not positive, we throw an exception.</p> <pre>int number; cout << "\n Enter a positive number. "; cin >> number; try //try block { if(number <=0) throw number; } catch(int i) // caught an exception { cout << "\n ERROR, your number is not positive."; }</pre>	C++ Ref
<i>typedef</i>	<p>The <i>typedef</i> defines new data type names. For example, it is possible to <i>typedef</i> int's as WHOLE_NUMBERS, as follows:</p> <pre>typedef int WHOLE_NUMBERS;</pre> <p>You can then declare variables of type WHOLE_NUMBERS like this:</p> <pre>int main() { WHOLE_NUMBERS x, y, z;</pre> <p>The <i>typedef</i> is a required statement in C code (not C++) to define a structure tag as a data type. (This statement is not required in a C++ compiler!)</p> <pre>struct MyStruct { // struct members }; typedef MyStruct MYSTRUCT;</pre> <p>Then use MYSTRUCT to declare structure variables.</p>	C++ Ref
<i>typeid</i>	Provides the type of an object during program execution. Must include TYPEINFO.H header.	C++ Ref
<i>typename</i>	The <i>typename</i> keyword has two purposes. First, it can be substituted for the keyword <i>class</i> in the template declaration. Second, it tells the compiler a name is a type name instead of an object name.	C++ Ref

<i>union</i>	A declaration for memory location that is shared by two or more different variables or variable types at different times.	C++ Ref
<i>unsigned</i>	A data type modifier for integers and chars. An <i>unsigned</i> variable does not contain negative values. Can be used with char and int.	2
<i>using</i>	A directive statement used with namespace objects. The <i>using</i> statement tells which namespace (or region of the program) that the compiler must read. <code>using namespace std;</code> See <i>namespace</i> .	C++ Ref
<i>virtual</i>	The <i>virtual</i> keyword is used in function declarations in base classes for functions that will be redefined in inherited classes. <code>class Base { public: virtual void WhatAmI(); }; class Derived : public Base { public: void WhatAmI(); // actual function implementation foundhere };</code>	8
<i>void</i>	A data type that represents "nothing," such as in a function prototype. Here the function FUNC does not return anything; therefore, it has a <i>void</i> return type. <code>void FUNC();</code>	2
<i>volatile</i>	An access modifier for variables whose value may be changed by ways not specified by the program.	C++ Ref
<i>wchar_t</i>	A data type declaration for variables that hold wide characters. Wide characters are needed in character sets of languages that have more than 255 characters.	C++ Ref
<i>while</i>	A loop/iteration control statement. The <i>while</i> loop continues to execute as long as the condition remains true. <code>int count; while (count < 10) { cout << "\n Counting " << count; ++ count; }</code>	3



Appendix C

Operators in C++

The operator precedence (or priority) of operations is listed in Table C-1 in descending order. The highest priority is at the top of the list and the lowest priority is at the bottom. The associativity refers to the position of the operator in the C++ statement, not in this table. For example, in the following statement, the division will be performed before the multiplication because the associativity for these two operators is left to right. The addition is performed after the division and multiplication.

```
x = a / b * c + t;      // division, multiplication, addition, assignment
```

TABLE C-1
Operators in C++

Type	Symbol	Associativity
Resolution	::	Left to right
Primary	() [] .-> casting operators	Left to right
Unary	++ - & * ! ~ sizeof (type)	Right to left
Arithmetic	* / %	Left to right
Arithmetic	+-	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Relational	== !=	Left to right
Bitwise	& (AND)	Left to right
Bitwise	^ (XOR)	Left to right
Bitwise	/ (OR)	Left to right
Logical	&& (AND)	Left to right
Logical	(OR)	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= etc.	Right to left
Comma	,	Left to right

ASCII Character Codes



Appendix D

Table D-1 shows the ASCII character codes for decimal 0 to 127 (a few of the codes do not have visible symbols). Following Table D-1 are two short C++ programs and images of the output they produce. Program D-1 writes out the decimal, hex, octal, and symbol for codes 1–31. The output is seen in Figure D-1. Program D-2 writes the ASCII codes for 128–255, and this output is seen in Figure D-2. You can use these images as guides for writing graphical characters in your *cout* statements. Also, Visual C++ 2005 Express Help provides a complete description of the 255 ASCII codes, including the control sequences. Refer to the MSDN Help included with your Visual C++ 2005 Express software. Find the index listing for “ASCII Character Codes.”

TABLE D-1
ASCII Character Codes for 0 to 127

Decimal	Hex	Octal	Symbol
0	0	0	NULL
1	1	1	☺
2	2	2	(See Figure D-1)
3	3	3	♥
4	4	4	♦
5	5	5	♣
6	6	6	♠
7	7	7	Beep or bell
8	8	10	Backspace
9	9	11	Horizontal tab
10	A	12	\n newline
11	B	13	◊ vertical tab
12	C	14	♀/ form feed
13	D	15	↵ Enter key or carriage return
14	E	16	(See Figure D-1)
15	F	17	(See Figure D-1)
16	10	20	(See Figure D-1)
17	11	21	(See Figure D-1)
18	12	22	(See Figure D-1)

Continued

■ TABLE D-1

ASCII Character Codes for 0 to 127 (Continued)

Decimal	Hex	Octal	Symbol
19	13	23	(See Figure D-1)
20	14	24	(See Figure D-1)
21	15	25	(See Figure D-1)
22	16	26	(See Figure D-1)
23	17	27	(See Figure D-1)
24	18	30	(See Figure D-1)
25	19	31	(See Figure D-1)
26	1A	32	(See Figure D-1)
27	1B	33	Escape
28	1C	34	(See Figure D-1)
29	1D	35	(See Figure D-1)
30	1E	36	(See Figure D-1)
31	1F	37	(See Figure D-1)
32	20	40	Space
33	21	41	!
34	22	42	"
35	23	43	#
36	24	44	\$
37	25	45	%
38	26	46	&
39	27	47	,
40	28	50	(
41	29	51)
42	2A	52	*
43	2B	53	+
44	2C	54	/
45	2D	55	-
46	2E	56	.
47	2F	57	/
48	30	60	0
49	31	61	1
50	32	62	2
51	33	63	3
52	34	64	4
53	35	65	5
54	36	66	6
55	37	67	7
56	38	70	8
57	39	71	9
58	3A	72	:
59	3B	73	;
60	3C	74	<

61	3D	75	=
62	3E	76	>
63	3F	77	?
64	40	100	@
65	41	101	A
66	42	102	B
67	43	103	C
68	44	104	D
69	45	105	E
70	46	106	F
71	47	107	G
72	48	110	H
73	49	111	I
74	4A	112	J
75	4B	113	K
76	4C	114	L
77	4D	115	M
78	4E	116	N
79	4F	117	O
80	50	120	P
81	51	121	Q
82	52	122	R
83	53	123	S
84	54	124	T
85	55	125	U
86	56	126	V
87	57	127	W
88	58	130	X
89	59	131	Y
90	5A	132	Z
91	5B	133	[
92	5C	134	\
93	5D	135]
94	5E	136	^
95	5F	137	-
96	60	140	,
97	61	141	a
98	62	142	b
99	63	143	c
100	64	144	d
101	65	145	e
102	66	146	f

Continued

■ TABLE D-1

ASCII Character Codes for 0 to 127 (Continued)

Decimal	Hex	Octal	Symbol
103	67	147	g
104	68	150	h
105	69	151	i
106	6A	152	j
107	6B	153	k
108	6C	154	l
109	6D	155	m
110	6E	156	n
111	6F	157	o
112	70	160	p
113	71	161	q
114	72	162	r
115	73	163	s
116	74	164	t
117	75	165	u
118	76	166	v
119	77	167	w
120	78	170	x
121	79	171	y
122	7A	172	z
123	7B	173	{
124	7C	174	
125	7D	175	}
126	7E	176	~
127	7F	177	Delete

Program D-1

```

1 //Appendix D ASCII Character Codes
2 //Low Range 1-31
3 //File: AppDASCIICodes.cpp
4
5 #include <iostream> //needed for cout to screen
6 #include <iomanip> //for setw()
7 using namespace std;
8
9 int main()
10 {
11     int i=1,ctr;
12
13     cout<<"\n ASCII Character Codes 1-31 "

```

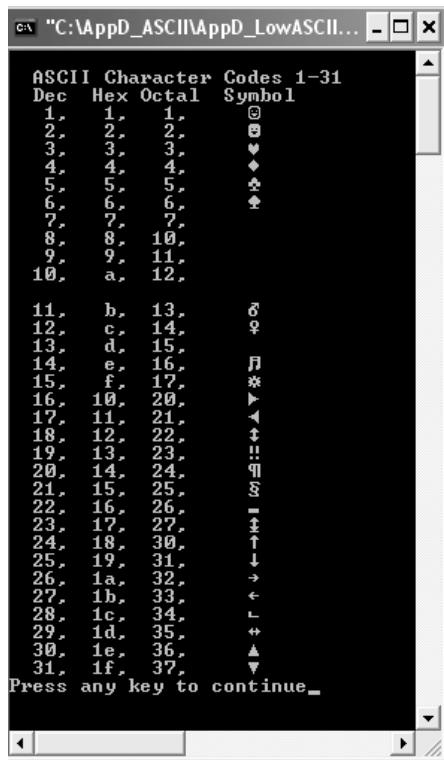


Figure D-1

The ASCII codes ranging from decimal 1-31.

```
14             << "\n Dec  Hex Octal  Symbol\n";
15     for(ctr = 1;ctr <= 31; ++ctr)
16     {
17         cout << setw(4) << dec << ctr << ",";
18         cout << setw(4) << hex << ctr << ",";
19         cout << setw(4) << oct << ctr << ",";
20         cout << setw(6) << char(ctr) << endl;
21     }
22     return 0;
23 }
24 }
```

Program D-2

```
1 //Program prints out the characters for
2 //ASCII codes 128-255
3
4 #include <iostream>
5 #include <iomanip>
6 using namespace std;
7
```

```
8  int main()
9  {
10     //print 9 int/char pairs on each row
11
12     int count, decimal = 128;
13
14     cout << "\n ASCII Character Codes 128-255 \n\n";
15     while(decimal < 256)
16     {
17         count = 0;
18         while(count < 9)
19         {
20             cout << setw(5) << decimal <<
21                 setw(3) << static_cast<char>(decimal);
22                                         ++decimal;
23                                         ++count;
24                                         if(decimal == 256)break;
25         }
26         cout << endl << endl;
27     }
28     cout << endl;
29     return 0;
30 }
31
```

The screenshot shows a Windows command-line window titled "C:\AppD_ASCII\appD_HighASCII\Debug\appD_HighASCII.exe". The window displays a table of ASCII character codes from 128 to 255. The columns represent the decimal code, the character itself, and its corresponding ASCII code. The table is as follows:

ASCII Character Codes 128–255																	
128	ç	129	ü	130	é	131	â	132	ä	133	à	134	å	135	ç	136	ê
137	ë	138	è	139	ï	140	î	141	ì	142	ñ	143	ß	144	É	145	æ
146	Œ	147	â	148	ö	149	ò	150	û	151	ù	152	ÿ	153	ö	154	ü
155	¢	156	£	157	¥	158	฿	159	ƒ	160	á	161	í	162	ó	163	ú
164	ñ	165	Ñ	166	¤	167	¤	168	¸	169	⌐	170	⌐	171	⌐	172	⌐
173	⌐	174	«	175	»	176	⌐	177	⌐	178	⌐	179	⌐	180	⌐	181	⌐
182	⌐	183	⌐	184	⌐	185	⌐	186	⌐	187	⌐	188	⌐	189	⌐	190	⌐
191	⌐	192	⌐	193	⌐	194	⌐	195	⌐	196	⌐	197	⌐	198	⌐	199	⌐
200	⌐	201	⌐	202	⌐	203	⌐	204	⌐	205	⌐	206	⌐	207	⌐	208	⌐
209	⌐	210	⌐	211	⌐	212	⌐	213	⌐	214	⌐	215	⌐	216	⌐	217	⌐
218	⌐	219	⌐	220	⌐	221	⌐	222	⌐	223	⌐	224	⌐	225	⌐	226	⌐
227	⌐	228	⌐	229	⌐	230	⌐	231	⌐	232	⌐	233	⌐	234	⌐	235	⌐
236	⌐	237	⌐	238	⌐	239	⌐	240	⌐	241	⌐	242	⌐	243	⌐	244	⌐
245	J	246	⌐	247	⌐	248	⌐	249	⌐	250	⌐	251	⌐	252	⌐	253	⌐
254	⌐	255															

Press any key to continue

Figure D-2

The ASCII codes ranging from decimal 128 to 255.



Appendix E

Bits, Bytes, Memory, and Hexadecimal Notation

Overview

When C/C++ programs are compiled and executed, physical memory in the computer is reserved and used for the program's variables and other program components. It is important for the C++ programmer to understand a few details about computer memory and how it is addressed with hexadecimal notation. This information is especially helpful when learning about pointers and how they work.

Computer Memory and Disks

Computers store pieces of information in the form of bits, which are commonly referred to as 1s and 0s. Computer systems use several types of physical materials for storing and accessing data. Fundamentally, all these systems maintain the bits of data in one of two states, which are interpreted as 1s and 0s. For example, magnetic media such as floppy and hard disks are similar to old phonograph records, and the tracks are charged into positive or negative states. Random access memory (RAM) in personal computers is made up of computer chips that have digital logic with two distinct voltage states, often referred to as high and low. This two-state representation for data storage is known as binary. Table E-1 summarizes the two-state storage methods for computers.

Bits and Bytes

When data is read from a floppy disk, the positive and negative regions are interpreted as a series of 1s and 0s. This continuous stream of 1s and 0s is grouped into 8-bit sections, or bytes. Eight bits make up 1 byte of data, and 4 bits represent $\frac{1}{2}$ byte, or a nibble.

Byte Formats: ASCII and EBCDIC There are two standard conventions for interpreting the series of bits: the American Standard Code for Information Interchange (ASCII) and the Extended Binary Coded Decimal Interchange Code (EBCDIC). Java and other web friendly languages use unicode character sets. Unicode has adopted the ASCII code. The personal computer uses the ASCII notation

TABLE E-1

Computer Media

Type of Material	Where Is It?	How Does It Work?	Note
Magnetic material	Floppy disk, zip disks, computer hard disks	Disk head sets the polarity on the media	Very important to keep magnets away from disks and computers because they can remagnetize them.
Digital logic	Random access memory (RAM)	Electronic chips have voltage values set high or low	Data in RAM is lost if the power is turned off.

for interpreting bits, whereas mainframes and some minicomputers use the EBCDIC format. (Appendix D, “ASCII Character Codes,” presents a complete set of codes.)

Table E-2 shows how the same bit patterns have different meanings on two different systems. The hexadecimal notation for the bit pattern is presented and will be discussed in detail below. The *0x* prefix is used to indicate hex notation.

Hexadecimal Notation To read the series of 1s and 0s, computer scientists have adopted *hexadecimal (hex) notation* as a way to write and understand bit patterns in bytes. This *hex* notation provides an easy and efficient way to interpret bit patterns and is used when addressing computer memory. The C++ programmer will see hex notation when he writes out the address of a variable or the value of a pointer.

Memory Is in Hex All programmers should have an appreciation of the complex science underlying the computer programming languages. What the programmer needs to know is that computer memory is addressed and reported in hexadecimal notation. When the programmer examines addresses in C++ (in pointer variables), the addresses will be shown in hex. Program E-1 writes the addresses of two floating point values.

The variable num1 has the value 3.6 and its address is 0x0066FFD4. This address is eight hex digits. The *0x* is the computer’s way of indicating that this address is a hex value. These eight digits represent 4 bytes because it takes two hex digits to represent 1 byte of computer memory. Each address shown is 4 bytes. Is this material confusing? Let’s examine hexadecimal notation in more detail.

TABLE E-2

Byte Format Conventions

Bit Pattern	Hex Notation	In ASCII (on a PC)	In EBCDIC (on a Mainframe)
0101 0000	0x50	P	&
0110 0001	0x61	a	/
0100 1100	0x4C	L	<
0110 1111	0x6F	o	?

Base 10 Decimal, Base 2 Binary, Base 16 Hexadecimal We are all familiar with counting in decimal notation (base 10) because that notation is our normal counting method. The symbol *0* represents none and the symbol *1* represents one. We use the symbol *5* to represent the number of fingers on one hand. But when asked how many eggs in a dozen, we do not have one symbol to represent twelve. We need to use two of our symbols, a *1* and a *2*, to represent twelve. Ten different symbols are used for writing numeric values. These symbols are *0, 1, 2, 3, 4, 5, 6, 7, 8*, and *9*. Our counting scheme has an individual symbol for representing zero through nine, but when we need to represent ten objects, we must use a combination of our symbols. We write ten using *10* (we have started reusing our symbols).

```
//Program E-1 Writing out hex addresses for two numbers.  
#include <iostream>  
using namespace std;  
int main()  
{  
    float num1, num2;  
    cout << "\n Please enter 2 floating point values ";  
    cin >> num1 >> num2;  
    cout << "\n You entered " << num1 << " and " << num2;  
    cout << "\n The addresses in hex of the numbers are " <<  
        &num1 << " and " << &num2 << endl;  
    cout << "\n Not bad, huh?" << endl;  
    return 0;  
}
```

Output

```
Please enter 2 floating point values 3.6 8.2  
You entered 3.6 and 8.2  
The addresses in hex of the numbers are 0x0066FFD4 and 0x0066FDF0  
Not bad, huh?
```

Because computer data is stored in bits, we have only two symbols, *1* and *0*, with which to write all the information for the computer. (Our alphabet uses twenty-six symbols to represent the English language.) All computer data is written in this two-symbol language known as *base 2* or *binary*. Since binary notation is so cumbersome, computer scientists use *base 16*, also known as *hexadecimal*, as a shorthand notation for representing the binary data. Hexadecimal notation uses sixteen separate symbols, *0* to *9* and *A, B, C, D, E*, and *F*. Table E-3 shows the different base representations for the numeric values *0* to *20*.

By combining the 8-bit byte and hexadecimal notation, computer scientists have a method where one digit (or symbol) can be used to represent 4 bits, and two hex digits (symbols) can be used to represent 1 byte of data. Decimal notation (*0* to *9*) would not be practical to use because it requires two digits to represent ten through fifteen.

Individual bit patterns can be obtained from hexadecimal notation. Two examples are shown in Table E-4. (Refer to Table E-3 for actual hex values shown in detail.)

TABLE E-3

Numeric Values and Different Bases

Numeric Amount	Base 2	Base 10	Base 16	Notes
Zero	0	0	0	
One	1	1	1	
Two	10	2	2	Base 2: start reusing symbols
Three	11	3	3	
Four	100	4	4	Base 2: we now use three symbols
Five	101	5	5	
Six	110	6	6	
Seven	111	7	7	
Eight	1000	8	8	
Nine	1001	9	9	
Ten	1010	10	A	Base 10: start reusing symbols
Eleven	1011	11	B	
Twelve	1100	12	C	
Thirteen	1101	13	D	
Fourteen	1110	14	E	
Fifteen	1111	15	F	
Sixteen	10000	16	10	Base 16: carry the 1 and start reusing symbols
Seventeen	10001	17	11	
Eighteen	10010	18	12	
Nineteen	10011	19	13	
Twenty	10100	20	14	Base 10: increment 1 to 2 and start the ones column at 0 again.

Counting in Hex It is not necessary for the C++ programmer to become an expert in hex arithmetic unless he is planning to do assembler-level programming or to work on programming hardware devices. It is important for the C++ programmer to be able to recognize hex notation and count in hex so that he can fully understand pointers and addresses.

TABLE E-4

Hex and Bit Patterns

Hex Pattern	Number of Bytes	Details	Bit Pattern
0xFF24	2	0xF = 1111, 0x2 = 0010, 0x4 = 0100 F F 2 4	1111 1111 0010 0100
0xFA1E	2	0xA = 1010, 0x1 = 0001, 0xE = 1110 F A 1 E	1111 1010 0001 1110

Learning to count in hex (base 16) is similar to learning to count in decimal (base 10). The difficult part is remembering that there are sixteen symbols in hex instead of the ten symbols in decimal. Let's practice counting in both base 10 and base 16. First, we will count from one to twenty-three in both base 10 and base 16:

Base 10 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23

Base 16 1,2,3,4,5,6,7,8,9, A, B, C, D, E, F,10,11,12,13,14,15,16,17

Now we'll count from 90 to 106 in base 10 and match these values to the comparable values in hex. Notice that in base 10, we run out of digits at ninety-nine and begin using a third digit.

Base 10 90,91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106

Base 16 5A,5B,5C,5D,5E,5F,60,61,62,63, 64, 65, 66, 67, 68, 69, 6A

Now we'll count from 250 to 259 in base 10. In the base 16 counting, we run out of digits at base 10 255 (0xFF) and begin using a third hex digit for base 10 256 (100).

Base 10 250,251,252,253,254,255,256,257,258,259

Base 16 FA, FB, FC, FD, FE, FF,100,101,102,103

Data Variables and Hex In Chapter 5, many sample programs showed memory locations and their corresponding hex addresses. With pointers it is helpful to have a minimal understanding of hexadecimal notation and memory. A C++ program must reserve memory to store the variables and know the location (or address) of each variable. This location or address is in hexadecimal notation.

The programmer need not worry about where in memory the variables are stored because that is the job of the operating system. However, it is convenient to know that each floating point variable requires 4 bytes of memory and each double requires 8 bytes, and when the programmer examines a variable's address (be it directly or in a pointer), it will be in hexadecimal notation.

Practice

In Figure E-1, we have five floating point variables, and the beginning address for the first variable is 0x0066FDE4. What are the other variables' starting addresses?

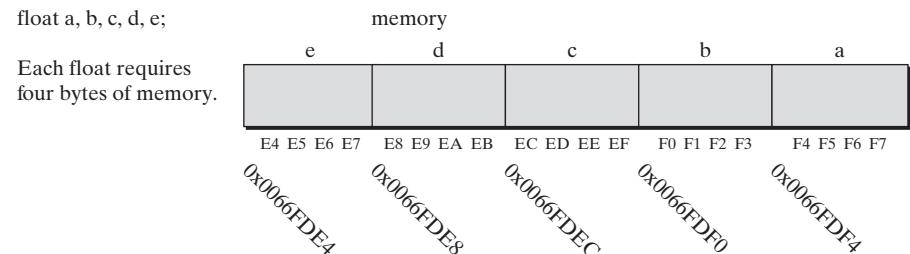


Figure E-1

Five floating point variables in memory.

Note that variables are stacked in memory from last to first. The last declared variable, “e,” is the first one in memory and it has the lowest address. Figure E-1 illustrates the computer memory.

The Most Common Mistake When Counting in Hex The most common mistake made when learning to count in hex is forgetting that there are sixteen symbols in hex. The order of the sixteen symbols (from smallest to largest) is shown here:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Always refer to base 10 counting as a guide and remember to carry one when you run out of symbols. When counting in hex, remember that the symbols go up to F instead of just 9. Table E-5 illustrates common mistakes made when counting in hex.

■ TABLE E-5

Right and Wrong Way to Count in Base 10 and Base 16 (Hex)

Technique	Count from Seven to Eighteen ^a
Right way: base 10	7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18
Right way: base 16	7, 8, 9, A, B, C, D, E, F, 10, 11, 12
Wrong way: base 16	7, 8, 9,  , 10,  , 1A, 1B, 1C, 1D, 1E, 1F, 20, 21
Technique	Count from Fourteen to Twenty-Seven
Right way: base 10	14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27
Right way: base 16	E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B
Wrong way: base 16	E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,  , 2A, 2B

^a  indicates incorrect counting sequence.





Appendix

F

File Input/Output

C++ provides several methods for reading and writing information from and to a data file. This appendix provides one technique for handling text-based data that is similar to reading data from the keyboard. The data in a text-based file is in ASCII format, and these data files can be created in a simple text-editor program (such as Microsoft’s Notepad). Before we present the C++ statements needed for reading and writing data files, we must understand data files in general.

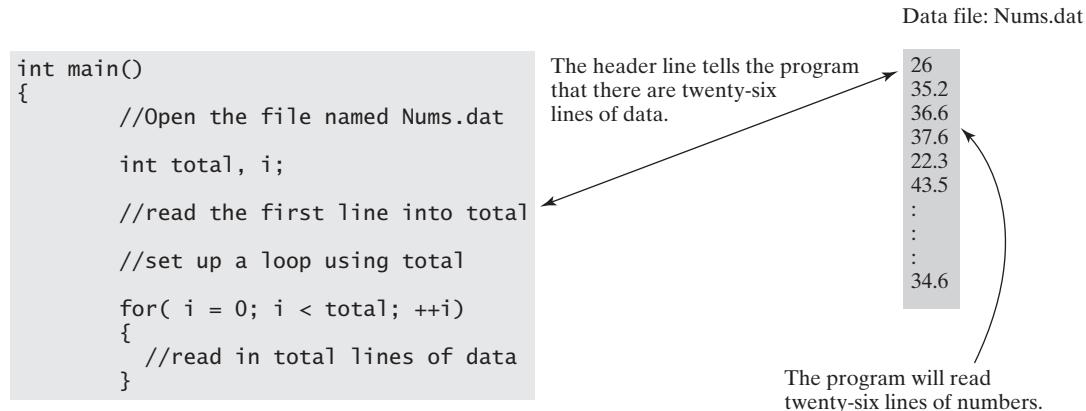
Know the File Format

When reading data from a file, you must know how the data is arranged in the file. This arrangement isn’t magic—it is part of the program design. Just as you decide how to organize your program and which functions to use, you also decide how the data files are to be formatted. Your program must have the read statements arranged so that they match the file layout, or your data will not be read correctly. Several general file format schemes are available.

Header Line The data file contains a header line(s) that relays information to the program concerning the data file. One example of this file type has an integer value in the first line. This value represents the number of lines following or other pertinent data. The program reads the first line into an integer variable and then uses it in a for loop. Figure F-1 shows the general flow of a program and a data file that has a header line and data.

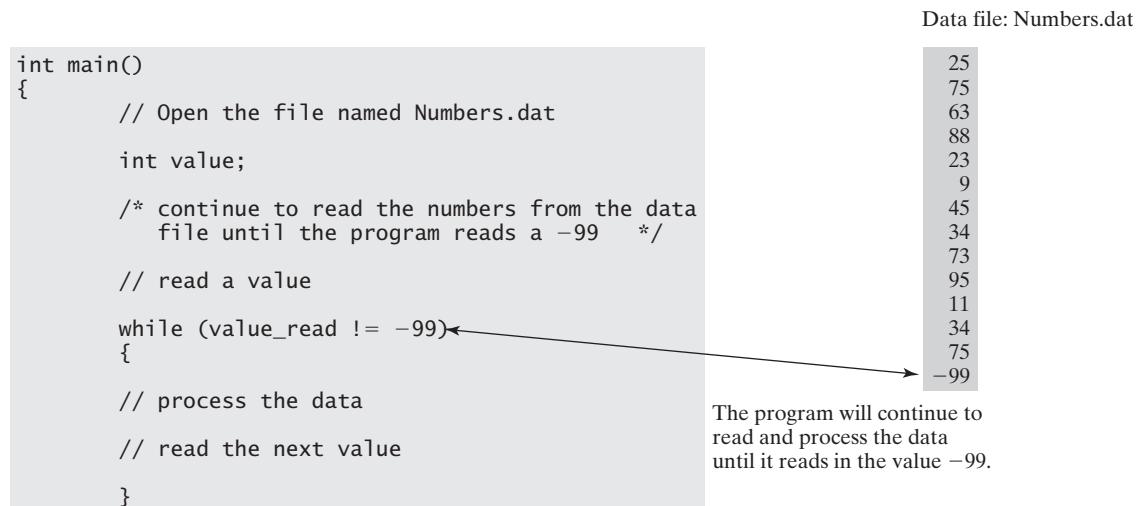
Trailer or Sentinel Another file format scheme involves placing a special value at the end of the data file and having the program check the values as they are read in from the file. When the program sees this special *trailer value* or *sentinel value*, it stops reading from the data file. Figure F-2 shows the program flow and data file in which the sentinel value is –99. The program continues to read the data file until it finds this value.

When using this technique, the programmer needs to pick a trailer value that is not within the range of the actual data. Also, if the program needs to read in several pieces of information on a line, such as a record of data, the programmer can designate one of the data items to be the trailer and trap for a special value.

**Figure F-1**

Header line data file scheme.

Read Until End of File A very common technique used when reading data files is to have the program read until it reaches the end of the file. When the input stream object opens a data file, it has a way to determine if it is at the end of the file. (One way to think of it is when C++ opens the file, it knows how many bytes of data are contained in the file. It keeps count of the number of bytes read, and hence will know when it is at the end of the file.) For our purposes, we'll ask the *ifstream* object if it is at the end of file, and use this EOF designation to our benefit. See Figure F-3.

**Figure F-2**

Trailer line data file scheme.

```

int main()
{
    // Open the file named Data.dat
    ifstream input;
    // open Data.dat

    int value;

    /* read the numbers from the data file until
       the program reaches the end of the
       file */

    while (! input.eof())
    {
        // read a value
        // process the data
    }

```

Data file: Data.dat

25
75
63
88
23
9
45
34
73
95
11
34
75

The program will continue to read and process the data until it reaches the end of the file.

The last entry is 75. It is at the end of the file.

Note: The value is typically read in the while's conditional statement.

Figure F-3

Read until the end of file.

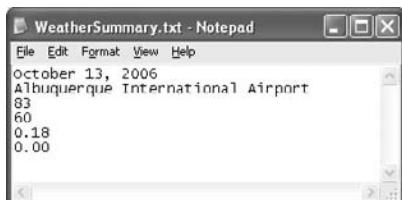
File I/O Using *iostream*

In C++, we originally introduced the data stream concept for input from the keyboard and output to the screen. We include the *iostream* library and use the *cin* and *cout* functions. When working with data files, we establish input and output stream objects and use these objects in the same manner as we used *cin* and *cout*. Open the stream object and use them exactly as we used our old pals *cin* and *cout*, and we're "in there" reading data from files!

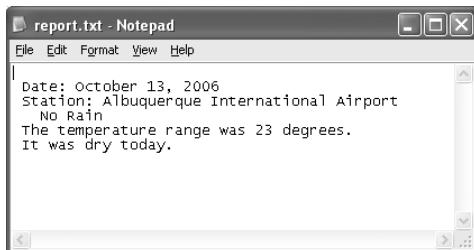
Steps for Text-Based File Input/Output

Step 1: Open the file. The program must first locate and open the data file. Files must be opened before data can be read. The required functions are in the *fstream* library. It is also convenient to use a *#define* statement near the top of your program so the file names can be found easily when reading source code. Assume that we will read data from the *WeatherSummary.txt* file and write information to the *Report.txt* file. In Program F-1, we set up the input and output stream objects and open the files, set the appropriate *ios* flag in the open statement, and check also that the input file is located. If the program cannot find the file, there is no point in continuing—so the standard library's exit function is used.

Step 2: Read the data into variables. Once the input and output stream objects have been created and opened successfully, the data can be read from the input file and written to the output file. To continue with this example, suppose the weather data input file was set up as shown in Figure F-4.

**Figure F-4**

The WeatherSummary.txt file contains weather data for Albuquerque.

**Figure F-5**

The report.txt output file.

We want our program to read in this data and report the temperature range, humidity (dry, nice, muggy), and whether or not it rained that day. We declare the appropriate variables and read the data line-by-line using the *ifstream* object, *input*. This works just like the *getline* for reading strings and *cin* for reading numeric data.

Step 3: Write output. To write data to the output file, we use the output stream in a manner similar to *cout*. (See code in Program F-1.) Figure F-5 shows the report.txt output file generated by this program.

Program F-1

```
1 //Program Weather Statistics
2 //File: AppFWeatherStats.cpp
3
4 #include <iostream>           //required for cout
5 #include <fstream>           //required for file I/O
6 #include <string>
7
8 using namespace std;
9
10 // place the input data file in the project folder
11 #define FILE_IN "WeatherSummary.txt"
12 #define FILE_OUT "report.txt"
13
14 int main()
15 {
16     //first must set up streams for input and output
17
18     ifstream input; //setup input stream object
19     ofstream output; //setup output stream object
```

```
20
21     //open for input
22     input.open(FILE_IN);
23
24     //Check file is opened.  Use the ! operator with input object.
25     if(!input )
26     {
27         cout << "\n Can't find input file " << FILE_IN;
28         cout << "\n Exiting program, bye bye \n ";
29         exit(1);
30     }
31
32     //Open the output file for the output
33     output.open(FILE_OUT);
34
35     //Now the two files are open and ready for reading
36     string date, station;
37     int high,low;
38     float humidity, rain;
39
40
41     //The read statements must match the data
42     //in the data file!
43
44     //Read the first 2 lines into strings
45     getline(input, date);
46     getline(input, station);
47
48     //Can use input like cin >> and read each
49     //line of numeric values into variables.
50     input >> high;
51     input >> low;
52     input >> humidity;
53     input >> rain;
54
55     //Now write the output file
56     output << "\n Date: " << date << "\n Station: "
57             << station;
58
59     if(rain == 0.0)
60         output << "\n    No Rain";
61     else
62         output << "\n    It rained today.";
63
64     int range = high - low;
65     output << "\n The temperature range was " << range
66             << " degrees. \n It was";
67
```



```
68     if(humidity < 0.33)
69         output << " dry ";
70     else if(humidity < 0.66)
71         output << " nice ";
72     else
73         output << " muggy ";
74
75     output << "today. " << endl;
76
77 //Now close the files.
78 input.close();
79 output.close();
80
81     return 0;
82 }
```

Example: Student Grades The file *StudentGrades.txt* contains student information on each line and is seen in Figure F-6. In Program F-2, we read in the data, calculate the average, and print the student's name and grade in the *Grades.txt* file. The student's grades are read into an array and we use the *FindAve* function. We need to declare array sizes. We write this program so that we read in eight grades for each student.

It is important to remember that the end of line characters must be handled correctly when one is reading data files. (Remember that we enter "\n" to tell *cout* to write to the next line, there is an end of line marker in the data file that is not visible.) If we have character data and numeric data and use a *cin*-like read for the numbers, the end of line marker must be pulled out of the input stream, as we do with *cin.ignore()* for the keyboard. (When reading just numeric data, as in the previous weather example cited above, the *cin*-like read statements ignore the whitespace characters.) Program F-2 illustrates the use of *getline* and character strings as well as the use of an individual *get* statement. Also, the data file is constructed carefully so that no extra spaces are accidentally placed at the end of the lines. The output data file is shown in Figure F-7. One important note, the data file is constructed so that the names are located in the first 24 characters on each line. We use a character array sized to 25 (24 chars plus the null) and we use the form of *getline* such that we tell it to read into the name array, and the 25 says, "Read up to 24 chars and put the null in the 25th element." We are then able to read the 8 numbers from the file using the *cin*-like properties of our *ifstream* object, *input*.

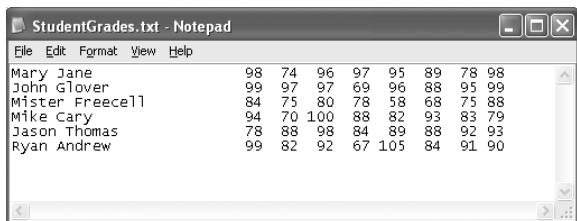
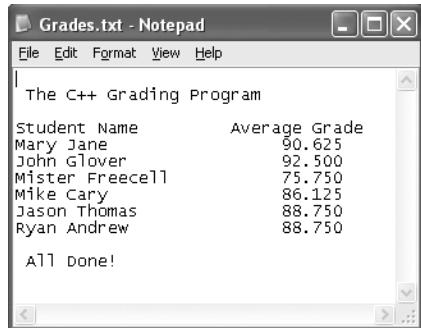


Figure F-6

The *StudentGrades.txt* file contains name and grade information for six students.

**Figure F-7**

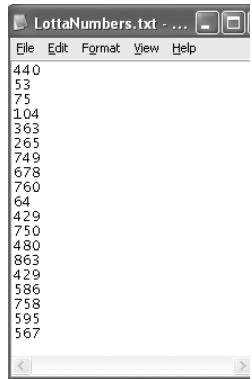
The Grades.txt file shows the results of our grading program.

Program F-2

```
1 // Calculate Student Grades
2
3 //File: AppFStudentGrades.cpp
4
5 #include <iostream>
6 #include <iomanip>           // required for setw
7 #include <fstream>           //required for file I/O
8 using namespace std;
9
10 #define FILE_IN "StudentGrades.txt"
11 #define FILE_OUT "Grades.txt"
12
13 float FindAve(float grades[], int total);
14
15 int main()
16 {
17     float grades[8], ave;
18     char name[25];
19     int total_grades = 8;
20
21     //first set up streams for input and output
22     ifstream input;
23     ofstream output;
24
25     //open for input
26     input.open(FILE_IN);
27
28
29     //Check to be sure file is opened.
30     if(!input )
31     {
32         cout << "\n Can't find input file " << FILE_IN;
33         cout << "\n Exiting program, bye bye \n ";
34         exit(1);
35     }
36
37     output.open(FILE_OUT);
```



```
38 //Write a header out to the output file
39
40     output << "\n The C++ Grading Program \n\n"
41         << "Student Name           Average Grade\n";
42
43     // Set the numeric output flags for
44     //3 decimal digits of precision
45     output.precision(3);
46     output.setf(ios::fixed);
47
48
49     //Now read each line of student data
50     //and calculate and print ave.
51     //Read until the end of file.
52
53     //use a char array to read 24 chars
54     while( ! input.eof())
55     {
56         input.get(name,25);      // read in the name
57
58         //use a for loop to read in the 8 grades
59         for(int j = 0; j < 8; ++j)
60         {
61             input >> grades[j];
62         }
63         input.ignore(); //pull out end of line char
64
65         ave = FindAve(grades,total_grades);
66
67         //now write date to output file
68         output << name << setw(8) << ave << endl;
69     }
70
71     output << "\n All Done! \n";
72     input.close();
73     output.close();
74     return 0;
75 }
76
77 float FindAve(float grades[], int total)
78 {
79     float sum = 0.0;
80     int i;
81     for(i = 0; i < total; ++i)
82     {
83         sum = sum + grades[i];
84     }
85     return(sum/total);
86 }
```

**Figure F-8**

The LottaNumbers.txt file contains several hundred integer values.

Ask for Filename and Read Until End of File The file *LottaNumbers.dat* contains a list of integer values. We do not know how many numbers are in this file, only that there is one number on each line. (See Figure F-8.)

In Program F-3 we ask the user to enter the name of the file. We read this in as a string, and then use the *c_str()* function to receive a character array needed for the open statement. The program opens the file and reads and adds these values (one at a time). Once again, we demonstrate how to continue reading a data file until we reach the end of file. The output is shown below the code.

Program F-3

```

1 //Ask for filename and read until EOF
2 //File: AppFAddAllThoseNumbers
3
4 #include <iostream>
5 #include <fstream>
6 #include <string>
7 using namespace std;
8
9 int main()
10 {
11     int total_numbers = 0, number, sum;
12     string filename;
13
14     //first must set up stream for input
15     ifstream ILoveNumbers;
16
17     cout << "\n Please enter the name of the file: ";
18     getline(cin,filename);
19
20     cin.ignore();
21
22     //filename is a string
23     //open needs a char []
24     //we use the c_str() function to give us

```

```
25 //the string in a char [] format
26 ILoveNumbers.open(filename.c_str());
27
28
29 //Check to be sure file is opened.
30 if(!ILoveNumbers )
31 {
32     cout << "\n Can't find input file " << filename;
33     cout << "\n Exiting program, bye bye \n ";
34     exit(1);
35 }
36
37 sum = 0;
38 while(!ILoveNumbers.eof() )
39 {
40     ILoveNumbers >> number;
41     sum = sum + number;
42     ++total_numbers;
43 }
44
45 cout << "\n           The file is " << filename;
46 cout << "\n Total Numbers in file " << total_numbers;
47 cout << "\n           The sum is " << sum;
48
49 cout << "\n Isn't that wonderful? Bye! " << endl;
50
51 ILoveNumbers.close();
52
53 return 0;
54 }
```



Output

```
Please enter the name of the file: LottaNumbers.txt
The file is LottaNumbers.txt
Total Numbers in file 250
The sum is 125024
Isn't that wonderful? Bye!
```

Binary File Input and Output: Overview

The secret to success with binary files is found in understanding pointers and addresses! (Are you surprised?) To *write* a binary file, we essentially open a file as binary and tell the program to write a chunk of data from memory into the file. We tell the *write* function the address of the data and how many bytes of memory to write. Then the binary data is packed into the file. When we *read* a binary file, we do just the opposite. We must provide the address of the memory where we wish the file data to be placed. To demonstrate, we write two

programs, Program F-4 writes data to a binary file, and Program F-5 reads and displays the data.

Individual data variables, arrays and objects, may be written into and read from binary files using this same write and read scheme. The pertinent code for writing a binary file is:

```
//first must set up streams for output
ofstream output;

//open it with the binary flag on
output.open(FILE_OUT, ios::out|ios::binary);

//Write blocks of chars to the file,
//use the & operator and sizeof() to indicate where
//where the data is and its size in bytes
output.write((char *)& VarName, sizeof(Data Type));
```

The input scheme is the same. One important note! You must read the data from the file in the exact order in which it was written!

```
//first must set up streams for input
ifstream input;

//open it with the binary flag on
output.open(FILE_IN, ios::in|ios::binary);

//Read blocks of chars to the file,
//use the & operator and sizeof() to indicate where
//where the data is and its size in bytes
input.read((char *)& VarName, sizeof(Data Type));
```

Examine Program F-4 and F-5. In both programs we have a simple Date class above the main function.

Program F-4

```
1 //Write to a Binary File
2 //File: AppFBinaryFiles.cpp
3
4 #include <iostream>
5 #include <fstream>
6 #include <string>
7 using namespace std;
8
9 #define FILE_OUT "BinaryData.dat"
10
11 class Date
12 {
13 private:
14     int month,day,year;
15 public:
16     Date(){month = day = 1; year = 2007; }
17     void SetData(int m, int d, int y)
```

```
18     {
19         month = m; day = d; year = y;
20     }
21     void Write()
22     {
23         cout << "\n Date: " << month << "/"
24             << day << "/" << year << endl;
25     }
26 };
27
28 int main()
29 {
30     Date d1, d2;
31
32     d1.SetData(1,10,1980);
33     d2.SetData(5,14,1970);
34
35     cout << "\n Write data to a binary file. \n";
36     cout << "\n Our dates are: ";
37     d1.Write();
38     d2.Write();
39
40     int numbers[5] = {53,73,92,23,21};
41
42     double PI = 3.14159265;
43
44     cout << "\n The array is " ;
45     for(int i = 0; i < 5; ++i)
46         cout << numbers[i] << " ";
47
48     cout.precision(10);
49     cout.setf(ios::fixed);
50     cout << "\n PI is " << PI << endl;
51
52     //first must set up streams for output
53     ofstream output;
54
55     output.open(FILE_OUT, ios::out|ios::binary);
56
57     //Write blocks of chars to the file.
58     output.write((char *) &d1, sizeof(Date) );
59     output.write((char *) &d2, sizeof(Date) );
60     output.write((char *) &numbers, sizeof(numbers) );
61     output.write((char *) &PI, sizeof(double) );
62
63     cout << "\n All done writing to the file! \n";
64     output.close();
65 }
```



```
66         return 0;  
67     }
```

Output

```
Write data to a binary file.  
Our dates are:  
Date: 1/10/1980  
Date: 5/14/1970  
The array is 53 73 92 23 21  
PI is 3.1415926500  
All done writing to the file!
```

Program F-5

```
1 //Read from Binary File  
2 //File: ReadFromBinaryFiles.cpp  
3  
4 #include <iostream>  
5 #include <fstream>           //required for file I/O  
6 using namespace std;  
7  
8 #define FILE_IN "BinaryData.dat"  
9  
10 class Date  
11 {  
12     private:  
13         int month,day,year;  
14     public:  
15         Date(){month = day = 1; year = 2007; }  
16         void SetData(int m, int d, int y)  
17         {  
18             month = m; day = d; year = y;  
19         }  
20         void Write()  
21         {  
22             cout << "\n Date: " << month << "/"  
23                 << day << "/" << year << endl;  
24         }  
25     };  
26  
27 int main()  
28 {  
29     Date d1, d2;  
30     int numbers[5];  
31     double PI;  
32  
33     cout << "\n Reading data from a binary file.\n";  
34     ifstream input,
```

```
36
37     //open for binary
38     input.open(FILE_IN, ios::in|ios::binary);
39
40     //Check to be sure file is opened. One way, use the fail
41     //function.
42     if(!input )
43     {
44         cout << "\n Can't find input file " << FILE_IN;
45         cout << "\n Exiting program, bye bye \n ";
46         exit(1);
47     }
48
49     input.read((char *) &d1, sizeof(Date) );
50     input.read((char *) &d2, sizeof(Date) );
51     input.read((char *) &numbers, sizeof(numbers) );
52     input.read((char *) &PI, sizeof(double) );
53
54     cout << "\n The dates are: ";
55     d1.Write();
56     d2.Write();
57
58     cout << "\n The array is " ;
59     for(int i = 0; i < 5; ++i)
60         cout << numbers[i] << " ";
61
62     cout.precision(10);
63     cout.setf(ios::fixed);
64     cout << "\n PI is " << PI;
65
66     input.close();
67
68     cout << "\n\n WOW! Most amazing file feats! " << endl;
69
70     return 0;
71 }
72
```



Output

Reading data from a binary file.

The dates are:

Date: 1/10/1980

Date: 5/14/1970

The array is 53 73 92 23 21

PI is 3.1415926500

WOW! Most amazing file feats!



Appendix G

Partial C++ Class Reference

This appendix contains reference material for C++'s string, vector, and queue classes which are introduced in this text. The reader is encouraged to explore a complete C++ reference in order to see all of the classes available to them.

C++ String Class

The C++ string class is a very useful class that is designed to contain textual data. The C++ string class is contained in the `<string>` library. Tables G-1 through G-5 present constructor and function information for the C++ string class.

TABLE G-1

String Class Constructors

Constructor	Job	Example
<code>string();</code>	Creates an empty string.	<code>string S1;</code>
<code>string(const char *str);</code>	Creates a string object from a null-terminated character array. It provides a conversion from a class C character array to a string. It is also used to create and initialize a string object.	<code>// conversion from char array char MyCString[25] = "I Love C++"; string S2(MyCString) //S2 now contains "I Love C++" // create and initialize string S3("What a great day.");</code>
<code>string(const string &str);</code>	Creates a string from another string.	<code>string S1; S1 = "I love C++"; string S2(S1); // S2 now contains "I love C++"</code>

TABLE G-2String Class Overloaded Operators^a

Operator	Job	Example
=	assignment	<code>S1 = "What a great day.";</code> <code>S2 = "Do you love C++?";</code>
+	concatenate Can be used with C-strings and C++ strings.	<code>S3 = S1 + S2; //hooks S2 on end of S1</code> <code>// S3 now has "What a great day.Do you</code> <code>love C++?"</code> <code>char array[20] = "Hello!";</code> <code>S1 += array;</code> <code>//S1 now has "What a great day.Hello!"</code>
+=	concatenation and assignment	<code>S1+=S2; // hooks "Do you love C++?" onto</code> <code>"What a great day.Hello!"</code> <code>// S1 now contains "What a great</code> <code>day.Hello!Do you love C++?"</code>
==	same as	<code>// compares the strings, 1 if same, 0 if</code> <code>not if(S1== S2)</code>
!=	not same as	<code>// compares the strings,</code> <code>//1 if not the same, 0 if same</code> <code>if(S1 != S2)</code>
<	less than	<code>if (S1< S2)</code> <code>/* compares the string for alphabetical</code> <code>order based on the ASCII code, where A</code> <code>to Z is lower than a to z and A is less</code> <code>than B. (See Appendix D, "ASCII</code> <code>Character Codes.")*/</code>
<=	less than or same as	<code>if (S1 <= S2) // see<</code>
>	greater than	<code>if(S1>S2) // see<</code>
>=	greater than or same as	<code>if(S1>= S2) // see<</code>
[]	subscripting Allows an array of strings to be created.	<code>string S[10]; // creates an array of 10</code> <code>strings</code>
<<	output	<code>cout << S1;</code>
>>	input ^a	<code>cin>> S1;</code>

^a `cin` will read to first whitespace character and terminate the reading process.

TABLE G-3

Partial Listing of String Class Editing Functions

Function	Job	Example
<code>insert(start, string);</code>	Place the string into the invoking string object at the start index.	<code>string S1("It is a rainy day."); string S2("very"); S1.insert(7,S2); //S1 is now "It is a very rainy day."</code>
<code>replace(start, size, string)</code>	Replaces size number of characters of invoking string object at the start index.	<code>//S1 is "It is a very rainy day." // S3 is "sunny" S1.replace(13, 6, S3); // S1 is "It is a sunny day."</code>
<code>erase(start, size);</code>	Erases size number of characters from the invoking string object at the start index.	<code>S1.erase(8, 5); // S1 is "It is a sunny day."</code>

TABLE G-4

Partial Listing of String Class Search Functions

Function	Job	Example
<code>find(string,start);</code>	Returns the first occurrence (i.e., index position) of the string located within the invoking string object. The search begins at the start index. The <i>npos</i> is returned if no match is found. (<i>npos</i> is defined to be -1)	<code>string S1 = "one potato two potatoes"; string S2 = "potato"; int First_pos; First_pos = S1.find(S2,0); //First_pos is 4</code>
<code>rfind(string,start);</code>	Returns the index of the string located within the invoking string object searching in reverse order. The search begins at the start index. (Note that <code>string::npos</code> is defined in the string class as the length of the string.) The <i>npos</i> is returned if no match is found.	<code>int Last_pos; Last_pos = S1.rfind(S2,string::npos); //Last_pos is 15 //string::npos defined to be length of the string</code>

TABLE G-5

Miscellaneous String Class Functions

Function	Job	Example
<code>assign(char [])</code>	Copy the contents of the character array into the invoking string.	<code>string S1;</code> <code>char MyArray = "I love C++";</code> <code>S1.assign(MyArray);</code> <code>//S1 now contains "I love C++"</code>
<code>assign(string, start, number)</code>	The number of characters from the string will be assigned into the invoking string, beginning at the start index. This function is normally used when a partial string assignment is needed. Use the = to assign an entire string.	<code>string S1, S2("I want a new car.");</code> <code>S1.assign(S2, 7, 9);</code> <code>//S1 now contains "a new car"</code>
<code>at(i)</code>	Returns the character at index i.	<code>string S1 = "apples";</code> <code>char c = S1.at(3);</code> <code>//C contains 'l'</code>
<code>append(string, start, number)</code>	Appends the number of characters from the string to the invoking string, beginning at the start index.	<code>string S1("I like to eat");</code> <code>string S2("peanuts and popcorn");</code> <code>S1.append(S2, 10, 8);</code> <code>//S1 now contains "I like to eat popcorn"</code>
<code>size</code>	Returns the length of string.	<code>string S1 = "bananas are good";</code> <code>int len = S1.size();</code> <code>//len has 16</code>
<code>c_str()</code>	<code>c_str()</code> function to convert a string to a <code>const char*</code> . The <code>c_str()</code> function returns a pointer to a null-terminated string; however, you may not change the contents of this string. (C++ does not guarantee the contents if you change the value; hence it requires the <code>char*</code> to be a <code>const char*</code> .)	<code>string S1("I love C++");</code> <code>const char *Array;</code> <code>Array = S1.c_str();</code>

C++ vector Class

The C++ vector class is designed to hold data elements in a linear list. When the data is added to the vector, the data remains in that order. It is similar to an array, but unlike an array, the vector can grow and shrink as needed. It is contained in the C++ <vector> library.

Tables G-6 and G-7 present partial listings of constructor and function information for the C++ vector class. These examples show vectors with integer and string data types. Other data types and classes can be used too. See a complete C++ reference for details.

C++ queue Class

The C++ queue class models a First In First Out (FIFO) line. As with a vector, you must declare a queue object and include the data type or class of the items that will be in the queue. It is contained in the <queue> library. Table G-8 presents a partial listing of queue functions. Since there is only one constructor, we'll show the way we constructed the queue above the table and use it with the function examples.

■ TABLE G-6
vector Constructors

Constructor	Example
Create zero length integer vector.	<code>vector<int> vNums;</code> <code>vector<string> vText;</code>
Create 10-element int vector and string vector.	<code>vector<int> vNums(10);</code> <code>vector<string> vText(10);</code>
Create an int vector from an int vector.	<code>vector<int> vNums2(vNums);</code>

■ TABLE G-7
vector Functions (Partial List)

Function	Job	Example
<code>reference back();</code>	Returns a reference to the last element in the vector.	<code>int last;</code> <code>last = vNums.back();</code>
<code>reference</code> is the data type for the vector		
<code>void clear();</code>	Removes all elements from the vector.	<code>vNums.clear();</code>
<code>bool empty();</code>	Returns a true if the vector is empty, false otherwise.	<code>bool result = vNums.empty();</code>

<code>reference front();</code>	Returns a reference to the first element in the vector.	<code>int first;</code>
reference is the data type for the vector		<code>first = vNums.front();</code>
<code>void pop_back();</code>	Removes the last element from the vector.	<code>vNums.pop_back();</code>
<code>void push_back();</code>	Adds an element to the end of the vector.	<code>int value = 7;</code> <code>vNums.push_back(value);</code>
<code>size_type size();</code> (for our purposes, size_type is an int)	Returns the number of elements currently in the vector.	<code>int length;</code> <code>length = vNums.size();</code>

TABLE G-8

queue Functions (Partial List). We assume queue object is constructed like this: `queue<int> qNumLine;`

Function	Job	Example
<code>bool empty();</code>	Returns a true if the queue is empty.	<code>bool result = qNumLine.empty();</code>
<code>reference front();</code> reference is the data type for the queue	Returns the first item that is in the queue, but does not remove it from the queue.	<code>int first = qNumLine.front();</code>
<code>void pop();</code>	Removes the first (front) item from the queue.	<code>qNumLine.pop();</code>
<code>void push();</code>	Pushes the item onto the end of the queue. When you push an item, it adds it to the end of the queue.	<code>int item = 7;</code> <code>qNumLine.push(item);</code>
<code>size_type size();</code>	Returns the number of items in the queue	<code>//fill queue int howMany = qNumLine.size();</code>

NOTE: The `front()` returns a reference to the first element in the queue but does not remove it. The `pop()` does not return anything, but removes the first element from the queue.



Appendix H

Multifile Programs

C++ programs should be constructed so that the program components are organized and grouped together into individual files. In general, include files (named with a *.h extension) contain #define statements, necessary #include statements, function prototypes, and class declarations. Source files (named with a corresponding *.cpp extension) contain necessary #include statements and the source code for all functions. Ideally, the *.h and *.cpp file set should have the same name (and different extensions) and contain the necessary source code for one class. Often this setup is not practical, but the file-set program components must be related to each other.

Simple Example: How Old Are You?

To illustrate how to break a program into multiple files, we revisit Program 4-2, How Old Are You? This program is contained in one *.cpp file named AskNameAge-Func.cpp. It has four functions and their prototypes are located above main. The function bodies are below main. The single-file version of this program appears as Program H-1.

Program H-1 (Also Program 4-2)

```
1 //Simple Functions and How Old Are You?
2
3 #include <iostream>
4 #include <string>
5
6 using namespace std;
7
8 //Function prototypes (or declarations)
9 void WriteHello();
10 string AskForName();
11 int AskForAge();
12 void Write(string name, int age);
13
```

```
14 int main()
15 {
16     int age;
17     string name;
18
19     //Write a greeting
20     WriteHello();           //call, no inputs, no return value
21
22     //Ask for the user's name
23     name = AskForName();   //call, returns name
24
25     //Ask for age
26     age = AskForAge();    //call, value assign into age
27
28     //Write information
29     Write(name, age);    //call, pass name, age to Write
30
31     return 0;
32 }
33
34 //Function definitions follow main
35
36 //Write_Hello writes a greeting message to the screen
37 void WriteHello()
38 {
39     cout << "\n Hello from a C++ function!\n";
40 }
41
42 //AskForAge asks the user for age, returns age.
43 int AskForAge()
44 {
45     int age;
46     cout << "\n How old are you? ";
47     cin >> age;
48     cin.ignore(); //remove enter key
49     return age;
50 }
51
52 //AskForName asks for the user's name
53 string AskForName()
54 {
55     string name;
56     cout << "\n What is your name? ";
57     getline(cin, name);
58     return name;
59 }
60
61
62 //Write writes the name and age to the screen
63 void Write(string name, int age)
```

```

64  {
65    cout << "\n Hi " << name << "! You are "
66          << age << " years old. \n";
67 }

```

In breaking this program into multiple files, we place the function prototype into the Functions.h file and its associated functions into the Functions.cpp file. The *main* function is placed in its own file, named Driver.cpp. We have to include the Functions.h file in Driver.cpp because it needs to see the prototypes of the functions that it calls. Note: It is important to realize that you need to include what each file needs to have! You may repeat including *<iostream>* or *<string>* Just concentrate on what each file needs to see, and include the appropriate items. Figure H-1 shows an outline of how we have split this program into multiple files. Figure H-2 shows the Microsoft Visual C++ 2005 project with multifile source code. Figure H-3 shows the console window with our program executing correctly. Figure H-4 shows how the three source files are located in the HowOldAreYou project folder.

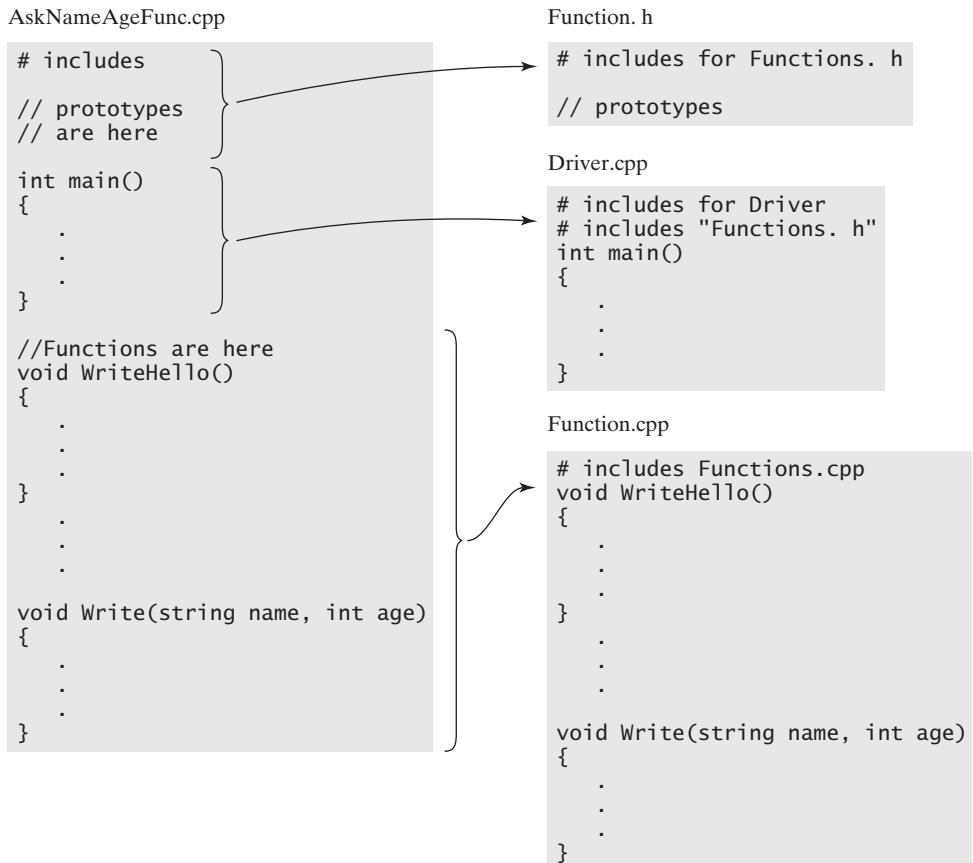
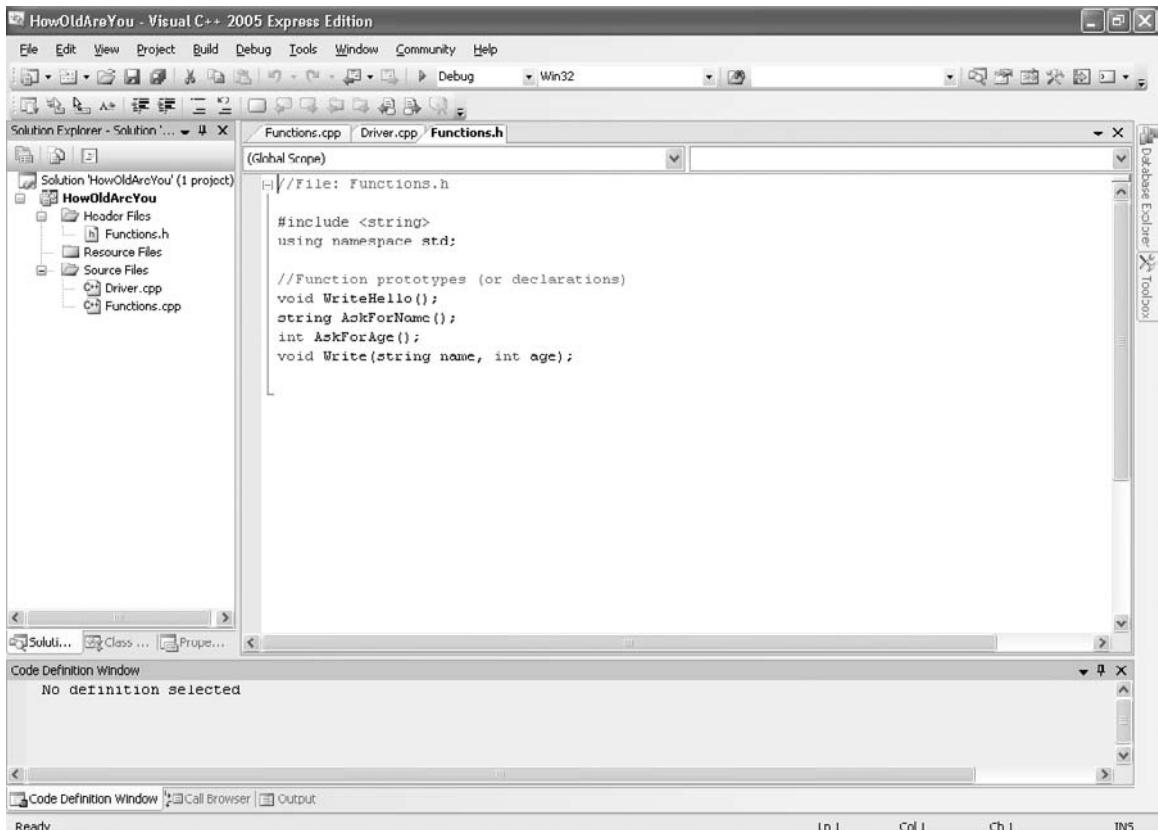


Figure H-1

This diagram shows how a single source file is split into multiple source files.

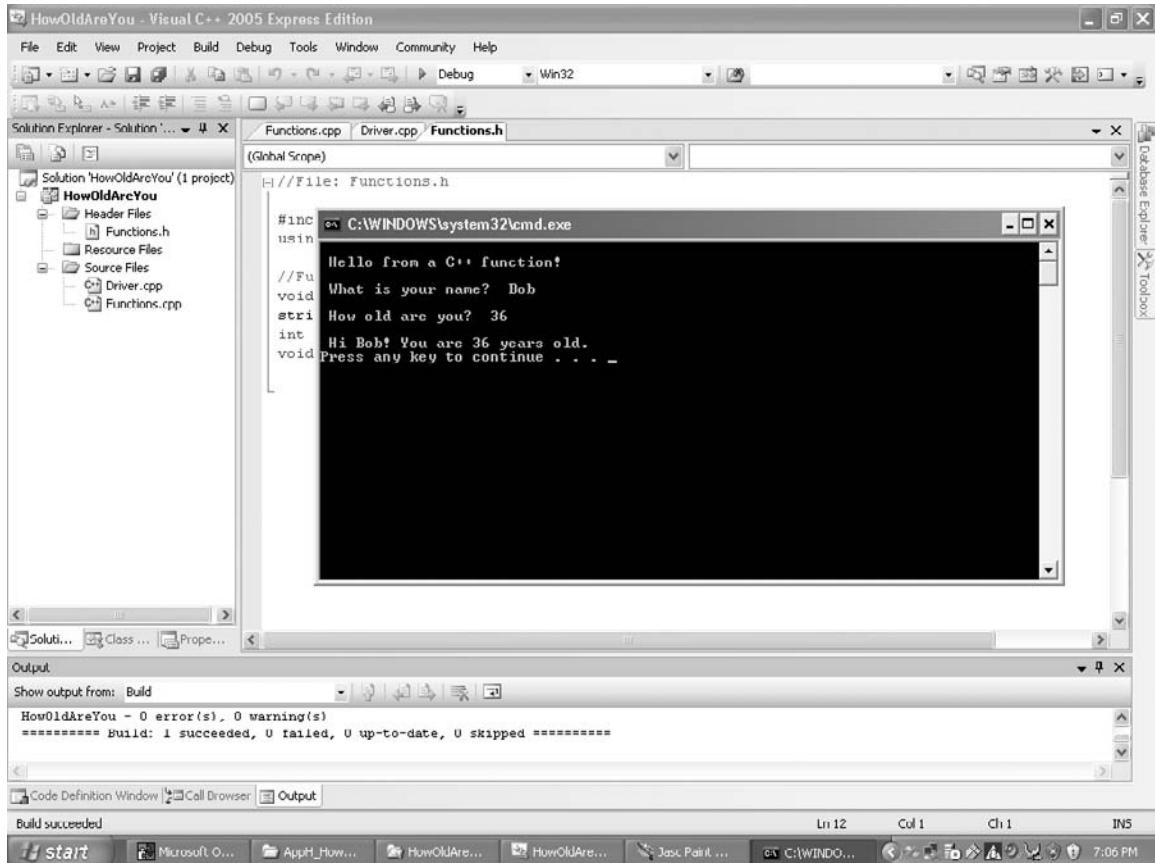
**Figure H-2**

The How Old Are You? program split into multiple source files in a Visual C++ 2005 project.

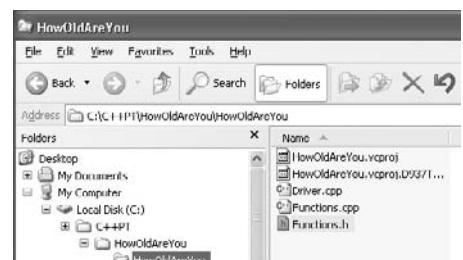
Program H-1

```
1 //File: Functions.h
2 //Prototypes for the How Old Are You? program
3 >
4 #include <string>
5
6 using namespace std;
7
8 //Function prototypes (or declarations)
9 void WriteHello();
10 string AskForName();
11 int AskForAge();
12 void Write(string name, int age);
13
```



**Figure H-3**

The HowOldAreYou program executes correctly in our multifile project.

**Figure H-4**

The three source code files are located in the HowOldAreYou project folder.

Program H-1

```

1 //File: Functions.cpp
2
3 #include <iostream>
4 #include <string>
5
6 using namespace std;
```

```
7 //Write_Hello writes a greeting message to the screen
8 void WriteHello()
9 {
10    cout << "\n Hello from a C++ function!\n";
11 }
12
13
14 //AskForAge asks the user for age, returns age.
15 int AskForAge()
16 {
17    int age;
18    cout << "\n How old are you?  ";
19    cin >> age;
20    cin.ignore(); //remove enter key
21    return age;
22 }
23
24 //AskForName asks the user's name
25 string AskForName()
26 {
27    string name;
28    cout << "\n What is your name?  ";
29    getline(cin,name);
30    return name;
31 }
32
33
34 //Write writes the name and age to the screen
35 void Write(string name, int age)
36 {
37    cout << "\n Hi " << name << "! You are "
38           << age << " years old.  \n";
39 }
```

Program H-1

```
1 //File: Driver.cpp
2
3 #include <iostream>
4 #include <string>
5 using namespace std;
6
7 //include for function prototypes
8 #include "Functions.h"
9
10 int main()
11 {
12    int age;
```



```
13     string name;
14
15     //Write a greeting
16     WriteHello();
17
18     //Ask for the user's name
19     name = AskForName();
20
21     //Ask for age
22     age = AskForAge();
23
24     //Write information
25     Write(name, age);
26
27     return 0;
28 }
```

Preprocessor Directives

Before we jump into the details of the `#include` statement, let's review the preprocessor directive. A preprocessor directive is a statement that provides instructions to the compiler. Each directive statement begins with a `#` and must be on its own line. There are twelve preprocessor directive statements in C and C++:

<code>#define</code>	<code>#endif</code>	<code>#ifdef</code>	<code>#line</code>
<code>#elif</code>	<code>#error</code>	<code>#ifndef</code>	<code>#pragma</code>
<code>#else</code>	<code>#if</code>	<code>#include</code>	<code>#undef</code>

Each directive serves a purpose. We require (and illustrate the use of) four of these statements (`#include`, `#define`, `#ifndef`, and `#endif`) when creating multifile programs. The reader should refer to a complete C++ reference for a full description and usage examples for all the preprocessor directive statements.

`#include` Statements and < > and " "

An `#include` statement tells the compiler to read another source file. In essence, it is saying, “Go read this file so that you know all of its contents.” The file that is in the `#include` line must be contained in either angle braces `< >` or quotes `" "`, such as:

```
#include <iostream>
#include "Functions.h"
```

A file enclosed in `< >` tells the compiler to look for the file in the language's standard location for include files. Buried deep in the Program Files folder on your computer, Microsoft Visual C++ 2005 places the language's include files in the

\ProgramFiles\Microsoft Visual Studio 8\VC\Include directory. Whenever Visual C++ encounters `<filename>` it looks in this directory automatically.

If a file is enclosed in double quotes, the compiler expects the full path for that file. If the file is located in the project folder (in Visual C++), no formal path description is necessary:

```
#include "Functions.h"
```

The compiler expects this Functions.h file to be located in the project folder. If the compiler does not find the file, it will not be able to find the functions that are prototyped and issue “unresolved external” messages.

Large programs often have all the include files located in a separate “Include” directory (which is a practical program design). In this case, the `include` statement must include the path, such as:

```
#include "\Includes\Functions.h"
```

It is possible to “nest” include statements. An include file can have an `#include` statement that contains a file with an `#include` statement (and so on). The number of layers of nesting that the ISO C language allows is eight, and the ISO C++ language provides for 256. Although 256 levels are possible, programmers are encouraged to nest their `#include` statements only a few layers deep.

What Do I Include Where?

One of the most confusing aspects of organizing a program with multiple files is knowing what needs to be included in what file. Beginning C++ programmers often take the shotgun approach and include everything in every file. This is a bad habit!

For each file in your program, the compiler must have the complete library and include information. For example, if you are writing output to the screen in a function, the file containing that function must have the `#include <iostream>` statement, even if you have already included it in another file. Look back at the How Old Are You? example. The iostream library is included in both the *Driver.cpp* and *Functions.cpp* files because both files contain a `cout` statement. Notice that the *Function.h* file does not contain the `#include <iostream>` statement. This file does not contain any `iostream` function calls—therefore, `iostream` is not needed.

As you are learning to put multifile programs together, you should ask yourself: What does this file need to know? If the file is performing a square root function, it needs C++’s math library and must have the `#include <cmath>` statement. If the file is referencing a class that is defined in *ClassName.h*, then you must include the *ClassName.h* file. It is common to have an include file included in several program files.

One last programming issue—you do not need to (nor should you include) *.cpp files within *.cpp files. It is the job of the project to keep track of your source files, and the project reads them automatically when the program is compiled.

#ifndef, #define, and #endif

It is important to understand that the compiler needs to read through a file only once during the compilation process. When the compiler sees an `#include` statement, it reads that file. If the compiler rereads the same file, it believes you are redefining something it has already read. (Compiler Errors!) This practice results in an unhappy compiler, which results in an unhappy programmer.

As our programs become more complicated, it is common that an include file is needed in several files. For example, when we write our Date class, it will be contained in the Date.h file. This Date.h file will need to be included in the Date.cpp file as well as wherever the Date object is needed. Therefore, conditional directive statements must be used in that include file. Basically, if an include file is to be included more than once, you must tell the compiler, “Compiler, if you haven’t read this yet, read it, but if you have read it, don’t read it again.”

Here is where the `#ifndef`, `#define`, and `#endif` directive statements come into play. Assume that we need to include the Date class in several files because several files will have Date objects. We need to wrap these statements around the prototype, as shown here:

```
// File: Date.h
#ifndef _DATE_H
#define _DATE_H

//class Date information

#endif
```

The `#ifndef` statement (if not defined) is saying to the compiler, “If you haven’t seen this item I’m calling `_DATE_H`, that is, if it isn’t defined yet....” (The compiler knows if it has seen the `_DATE_H`.) The `#define` line is telling the compiler, “Okay, define all the lines you read between this `#define` and `#endif` as `_DATE_H`.” The compiler then knows what `_DATE_H` is, and if it encounters another `#ifndef_DATE_H` statement, the compiler does not reread it.

Note that the `_DATE_H` label (actually it is a macro name) must be the same in both the `#ifndef` and the `#define` line. A C++ convention for this label uses all capital letters and underscores and the name should match the file name or give some indication about what it is. This idea is only a convention, not a requirement. If you examine some of the C++ library *.h files, you will see the following:

```
// The C library's math.h contains this set of statements
#ifndef _INC_MATH
#define _INC_MATH
```

Creating a Multifile Project in Microsoft Visual C++ 2005

This part is the easiest of the whole discussion. As you know, your Visual C++ Solution shows the Header Resource and Source Files folder in the window on the

left of the IDE. As we learned in Appendix A, you added your *.cpp file by right clicking on the Source folder and **Add→New Item**, selecting Code and C++ File(.cpp). When we start writing header files (*.h files), we select the Header File (.h) file. This adds the file to your project.

Once the files have been added to your solution, you may then double click on any of them and Visual C++ opens that file for your use. In theory you do not need to add the *.h files to the project (they are included in the source files). Figures H-5 and H-6 show the Visual C++ 2005 project workspace for Program 7-1, DateDemo from Chapter 7, and Program 8-3, Employee, Boss, and CEO from Chapter 8. The source and header files are seen in the left-hand window. In these images, you can see that we used the `#ifndef #define #endif` statements. Be sure that your files are organized and placed in the correct directories. As you can see, the multifile organization is extremely handy, especially if you have large C++ programs.

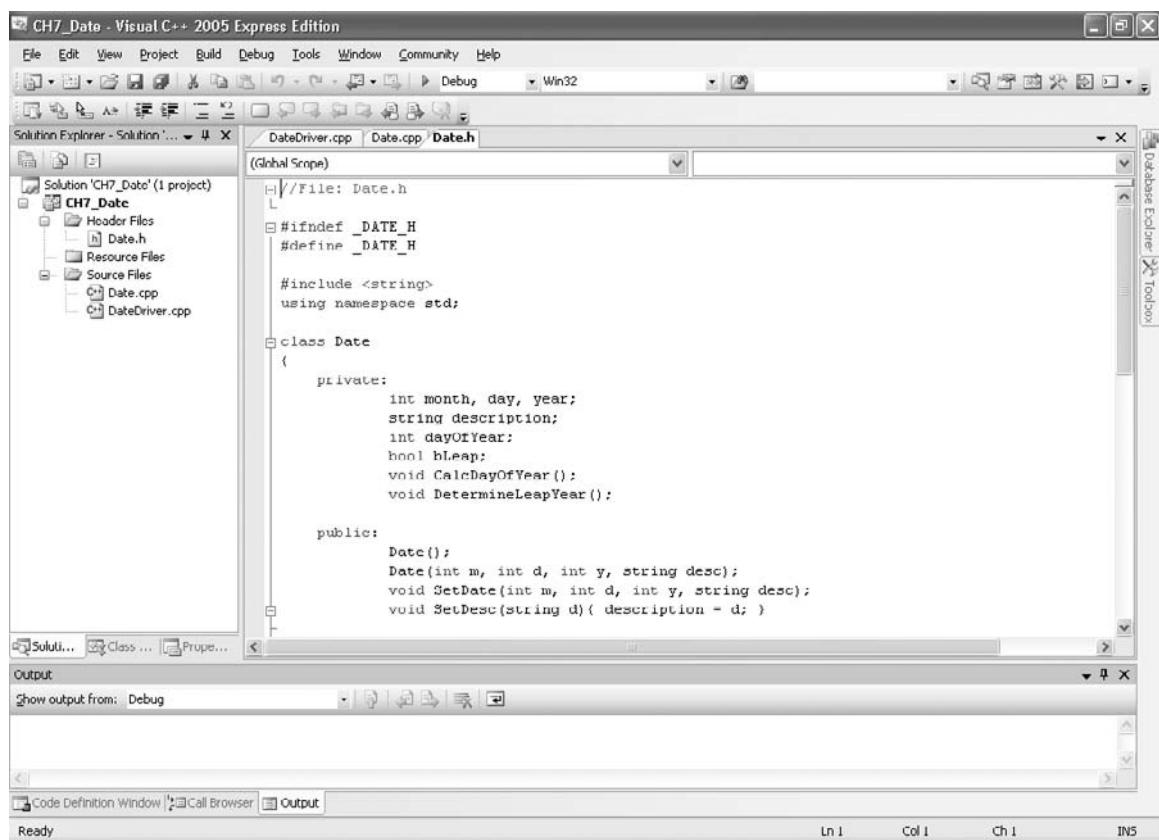
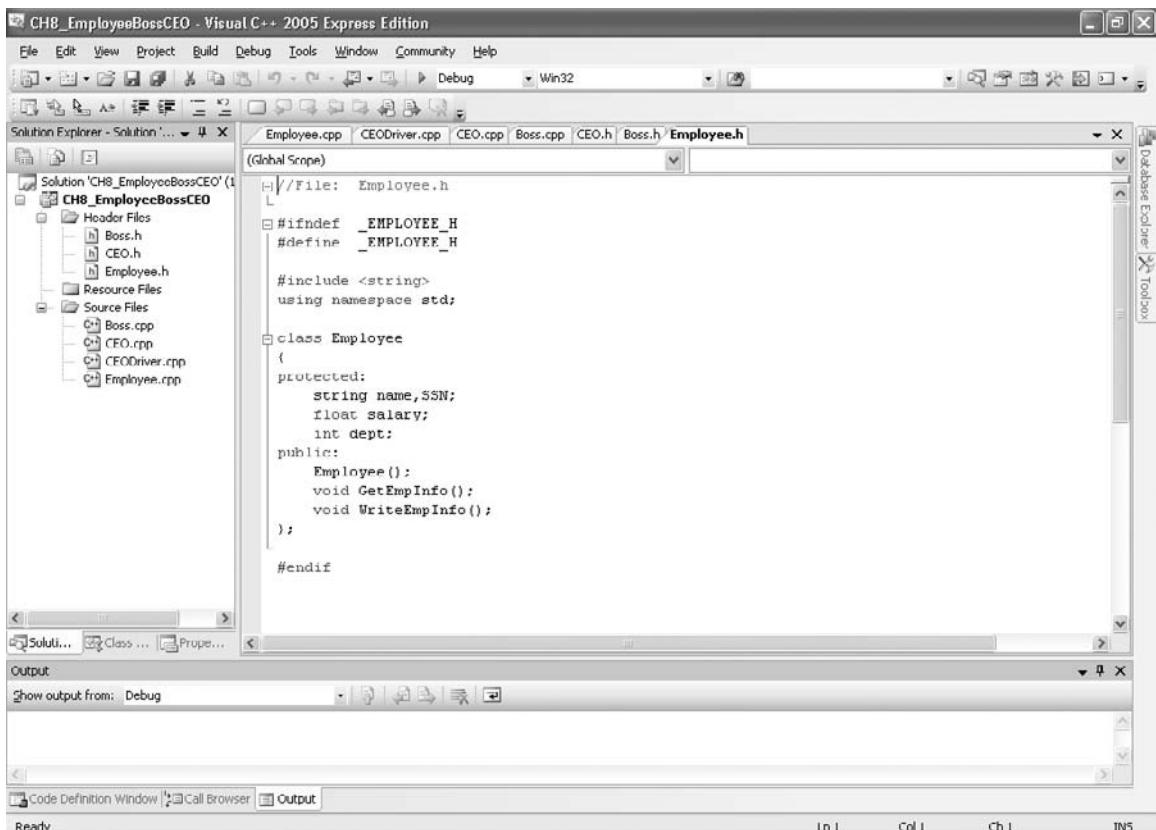


Figure H-5

The Date class program in Visual C++ 2005 multisource file project.

**Figure H-6**

The Employee, Boss, and CEO .cpp and .h files are well organized in multifile programs.

extern Global Variables in Multifile Projects

It is common in the structured programming world (and at times in the object-oriented world) to have variables declared in one file that are accessed by the program routines located in other files. In this situation, the variables must be declared globally and declared with the `extern` keyword in the second file.

Let's look at an example. Suppose a program contains a data variable, "total," and assume that our program needs to see the total in another routine located in a separate file. The "total" declaration looks like the following:

```
// File1.cpp
// Contains the original global declaration for total
double total;
int main()
{
    // code for calculating total
    total = //some magic formula
}
```

In the second source file, the total variable must be declared again with the `extern` keyword, like this:

```
// File2.cpp
// Contains the extern declaration for total
extern double total;
void Function1()
{
    // code that uses total
}
```

The `extern` statement basically tells the compiler, “Somewhere else in this program is a global declaration for a `double` named `total`.” Remember, there can be only one global declaration of a variable and there may be many `extern` declarations in many files.

extern Output Stream Example Suppose you wish to write your program output to a data file. Your program is set to have several source files, and it is logical to have information written from these program locations. One way to make the output stream object available to many program locations is to declare the output stream globally and then extern it in other program files. In the following short program, we declare the output stream object global to the `main` function and then extern it to the second file. The initial declaration is seen here:

```
//File: SayHelloMain.cpp
#include <iostream>
#include <fstream>
using namespace std;
ofstream Output;           //globally declared output stream object

void SayHelloToFile();
int main()
{
    Output.open("TestOutputFile.txt",ios::out);
    Output << "\n This is a test. This is written from main.\n";
    SayHelloToFile();
    return 0;
}
```

The output stream is written with the `extern` keyword in the file containing the `SayHelloToFile` function:

```
//File: SayHello.cpp
#include <iostream>
#include <fstream>
using namespace std;

extern ofstream Output;
void SayHelloToFile()
{
    Output << "\n\n Hello World in an output file.";
}
```





Microsoft Visual C++ 2005 Express Edition Debugger

Aside from knowing the language, problem, and software goals, knowing how to use the debugger is most important. The C++ programmer *must learn* how to use the debugging tools in whatever environment he is working.

The debugger allows you to step into your program and stop at any given line of code. You may run your program by stepping through it one line at a time and see the value for any and all variables in your program. In this manner, it is possible to follow exactly how the code executes and to see which lines are skipped. You write your software with certain expectations, and the debugger gives you the means to verify that the code is performing as anticipated—and to locate the point where logic or calculation failures occur.

Debugging Concepts

The Visual C++ debugger tool is powerful and easy to use. We should examine a list of new terms before we jump into an example. A complete listing and description of these tools is in the **Visual C++ Help → Index → Debugger** Window. Table I-1 covers the basic terms.

When your program compiles and runs but is not working correctly, use the debugger to examine how your program runs. You set one breakpoint (or several) in the program and then single-step through lines of your code, watching the variable values and seeing how the program flows. You may either step into functions or step over them. You may enter variables in the **Watch** window to monitor the program execution.

A Simple Example: Dumbbell Calculations from Chapter 2

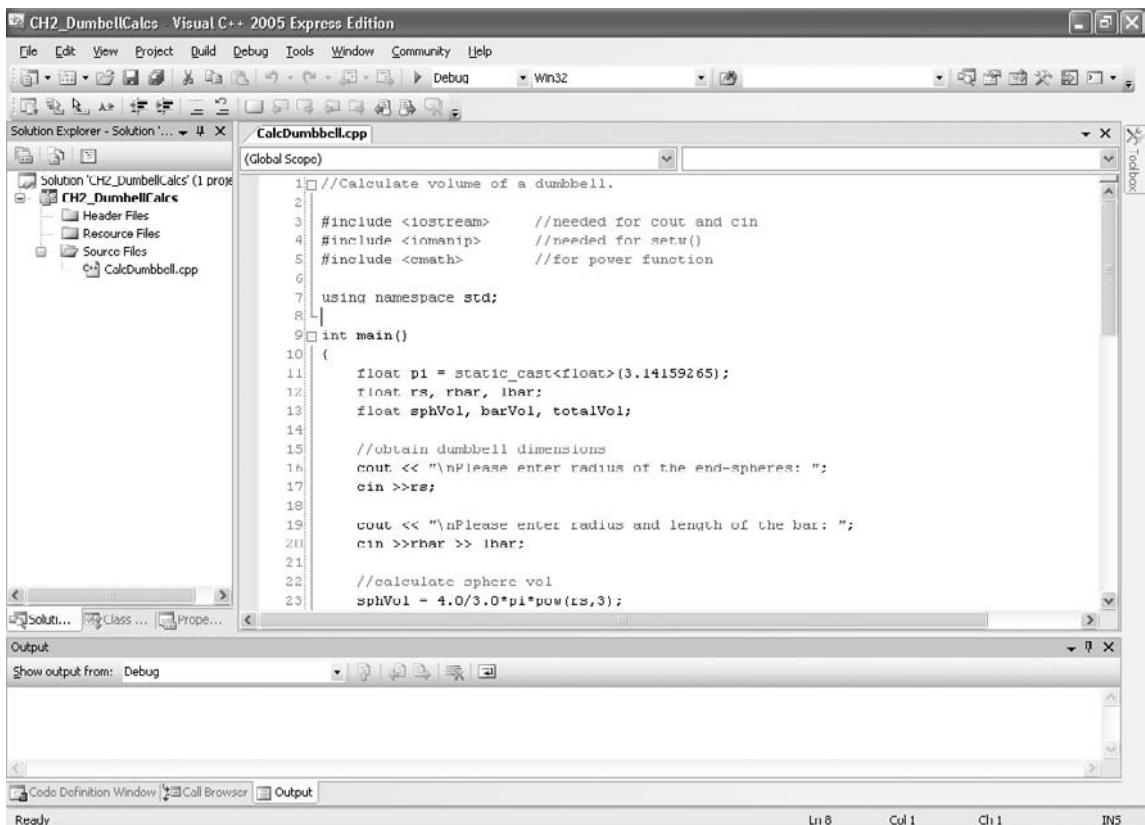
Let's use the Practice program in Chapter 2 to see how the debugger operates. Figure I-1 shows this project open and the start of the program. Let's turn on line numbers in our editor to make this discussion a bit easier. To do this, we follow this line down to a check box for line numbers. Here's the route we follow:

Tools—>**Options**—>**Text Editor**—>**All Languages**—>**General** and on the right is the Display choices. Turn on Line numbers.

We want to have the program stop at the sphere volume calculation line (line # 23). We put a breakpoint at this line by left clicking in the vertical gray bar beside the line number. We could also set the breakpoint by putting the cursor on the line and **Debug**—>**Toggle Breakpoint**, or hitting the F9 key. When a breakpoint is entered, you see the red dot on the left. (See Figure I-2.)

TABLE I-1
Visual C++ Debugger Terminology

Term	Menu/F Key Location	Use
Breakpoint	Debug —> Toggle Breakpoint or left click in bar left of source code or F9	A breakpoint is placed at a line of code in a program where you want the program to stop so that you may examine variables. Also, you may set a breakpoint where you wish to begin single-stepping through the code.
Go	Debug —> Start Debugging or F5	Start the debugger.
Stop	Debug —> Stop Debugging Shift F5	Stop the debugger.
Step into	F11	Step into a function. You can step along line-by-line in the function. Note: if you step into a C++ function, such as <i>cout</i> or <i>cin</i> , you will see the code in the iostream class.
Step over	F10	Step over a function. The function is executed but you don't see the line-by-line steps. You always want to step over <i>cout</i> , <i>cin</i> , <i>getline</i> , etc.
Step out	Shift F11	Step out of a function.
Run to cursor	Ctrl F10	Run the program and stop at the line where the cursor is located.
Windows	Debug —> Windows	Several debug windows are available for the program to select exactly what he wishes to see/follow as the program executes. The windows are shown below.
Local Window		Shows all of the variables that are declared locally within the currently executing function.
Watch Window		Allows you to enter different sets of watch variables. A watch variable is simply a variable that the programmer enters and then can watch its value continuously.
Auto Window		Shows the variables that are currently being part of the lines of code that are being executed.

**Figure I-1**

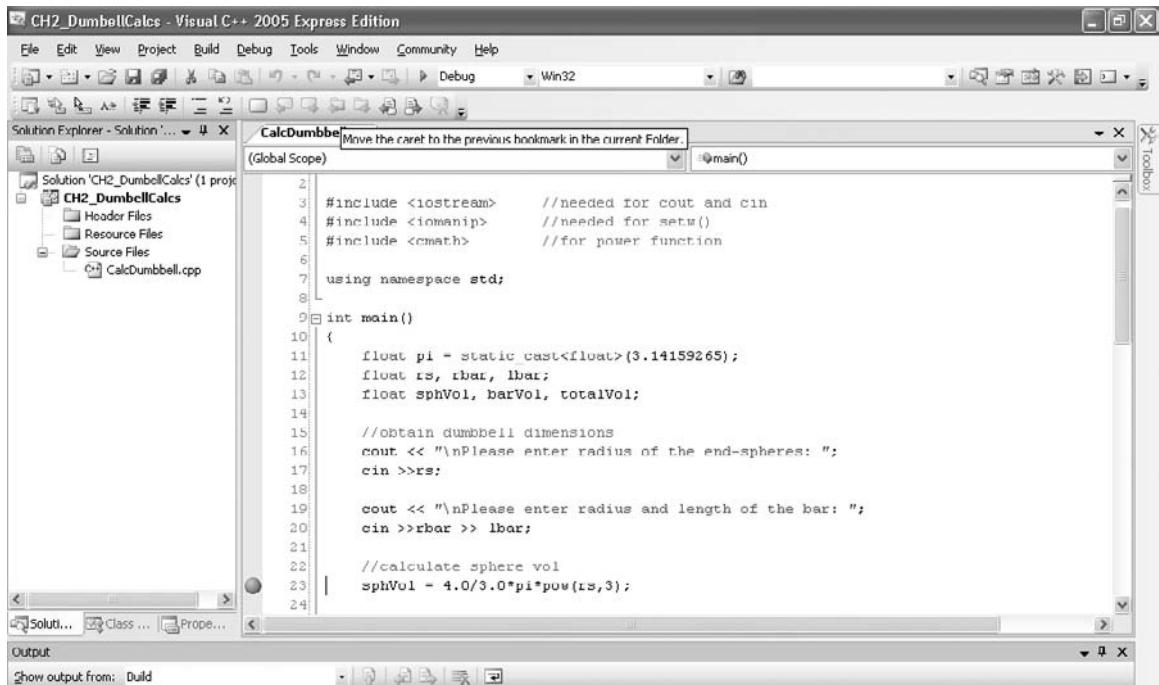
The Dumbbell Calculation Project open in Visual C++ 2005 Express with line numbering activated.

We'll start the debugger by pressing the F5 key. (We could have used the **Debug→Start Debugging** too. See Figure I-3.) The program shows the console window and we enter 5.0 for the radius, 1.0 for the bar radius, and 20.0 for the bar length. The program then stops at line 23. The yellow arrow is pointing at the line of code that will be executed next. By stopping the program at this point, we can see the values we've entered for the dumbbell dimensions in the lower left window. Notice how the sphVol value is a large negative number. This is because the program hasn't executed that line, and has not been initialized.

Now have the debugger step several lines by pressing F10—and you'll see the yellow arrow move to each line. Watch how each new value is shown in red in the lower left window. See Figure I-4.) It has calculated the sphVol, barVol and totalVol values. Our program is stopped on line 33, the yellow arrow indicated the next line the program will execute.

We can change the tab in the lower left corner to “Locals” and it will show us all the variables and their values in this *main* function. See I-4 again (Autos tab shows the variable involved in current calculations.)

To stop debugging, use the **Debug→Stop Debugging** or Shift F5 key.

**Figure I-2**

Breakpoint is seen as a red dot on the left side of the source code window.

Debugging Your Functions Step Into, Step Over

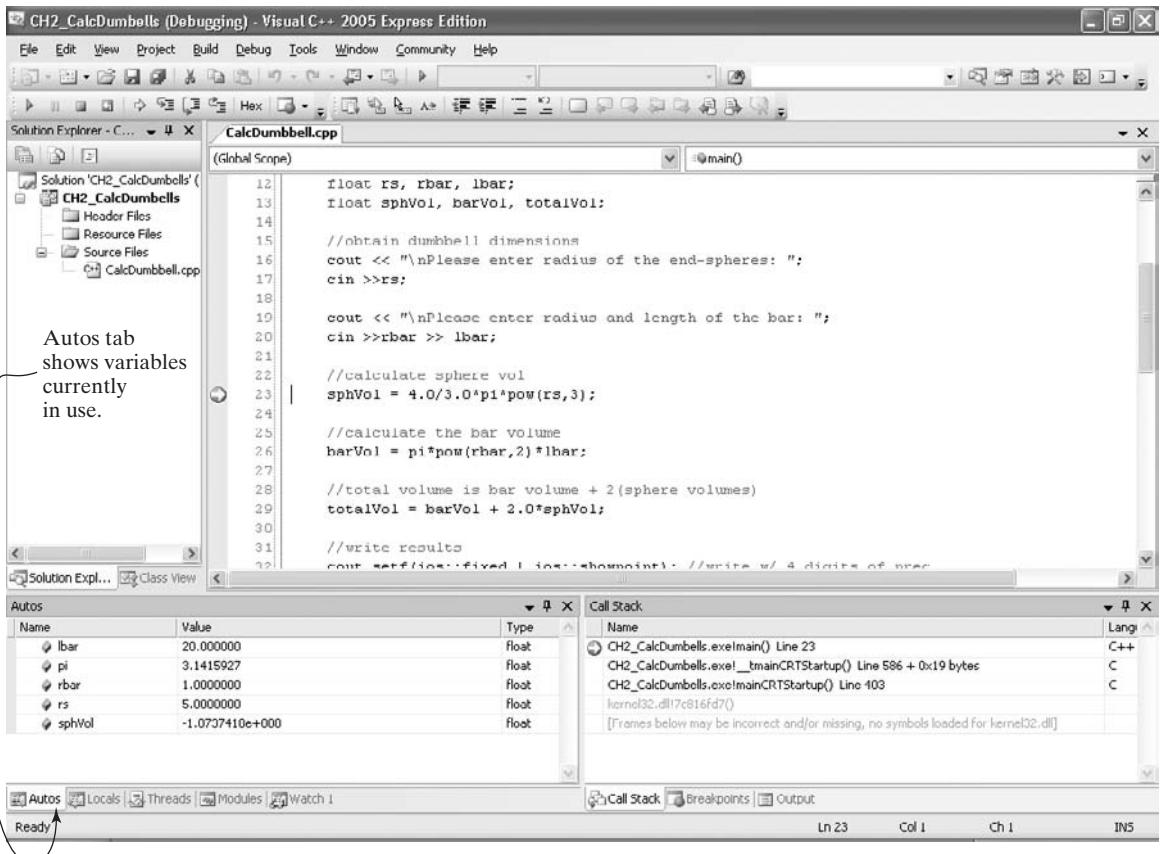
We'll now jump to Chapter 4 and use the *HowOldAreYou?* program to show how you can step into the functions that you write. We open our project and set a breakpoint at line 19, the call to the *AskForName()* function. (See Figure I-5, p.606.)

Pressing F5 starts the debugger, which stops at line 19. The program has executed the *WriteHello()* function, but not the *AskForName()*. (See Figure I-6, p.607.)

We want to now Step Into our function, so we hit the F11 key (or **Debug**—>**Step Into**). When we do this, we are now looking at line 26 of the *Functions.cpp* file, the first line of *AskForName()*. (See Figure I-7, p.608.)

We are able to hit the F10 key to step over (that is, execute the lines of code) and watch how this function executes. We can step out of the function too, pressing the Shift F11 key.

If we step along in our program, and enter "Bob" and "36", we can watch how the variables are passed back to *main*. We can then step into the *Write* function and see our data values have been passed into the function. (See Figure I-8, p.609.)

**Figure I-3**

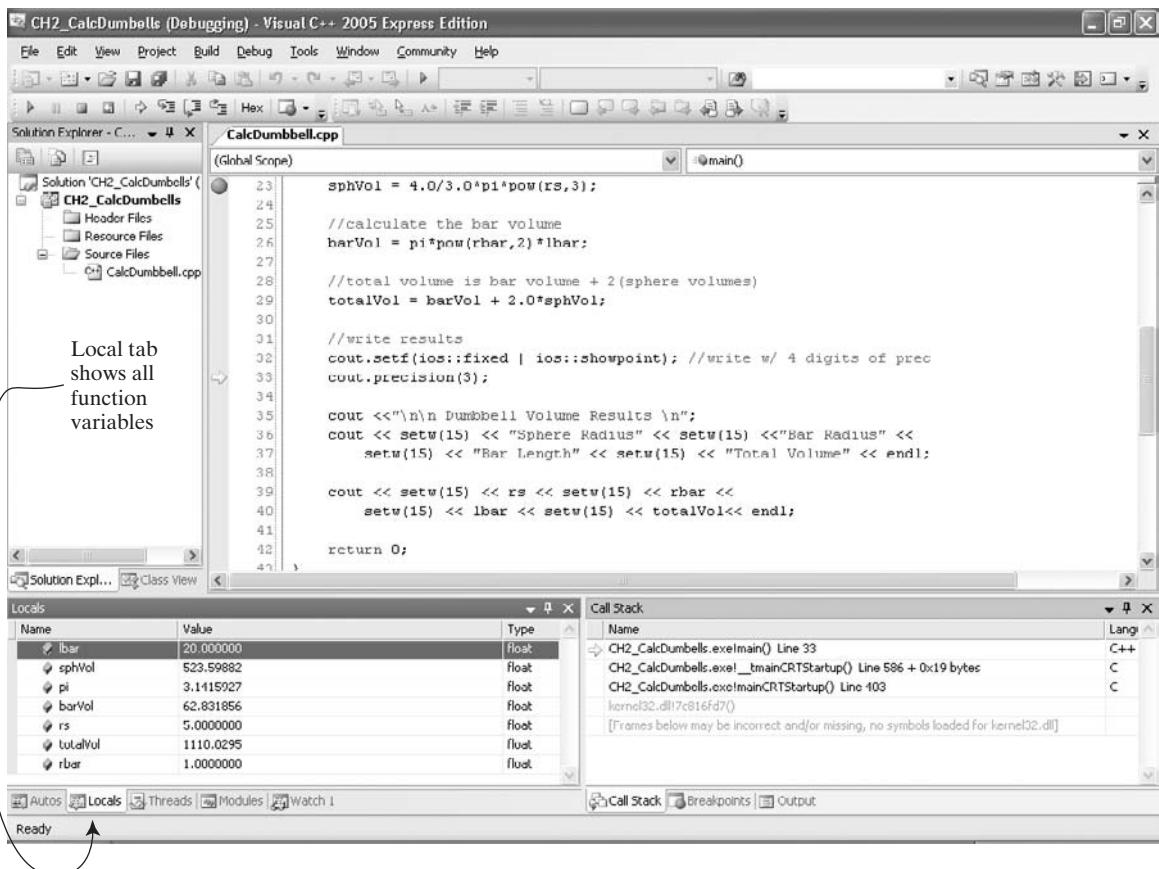
We've started the debugger and it has stopped at the breakpoint on line 23.

Debugging Classes

The Visual C++ 2005 debugger is an excellent way to see your class design as well as object values! Up to this point, we've used the Solution Tab showing our *.cpp and *.h files. In this last example, we bring up our DateDemo program from Chapter 7. We select the Class View tab and the *Date* class, and we see the class data and prototypes in the lower left pane. (See Figure I-9, p.610.)

This is very useful for you as a programmer, as it gives you a way to navigate easily through your source code. For example, when you click on the *GetFormattedDate* member, Visual C++ takes you to the start of that function. (See Figure I-10, p.611.)

We can see object data in the debugger too! In Figure I-11 we place a breakpoint at line 21 and then step to line 24. In this program, we have two *Date* objects, as well as two string objects. Notice how the lower left window shows a summary of the *Date* and string objects. Figure I-12 shows how we can see all details of the objects in an expanded view.

**Figure I-4**

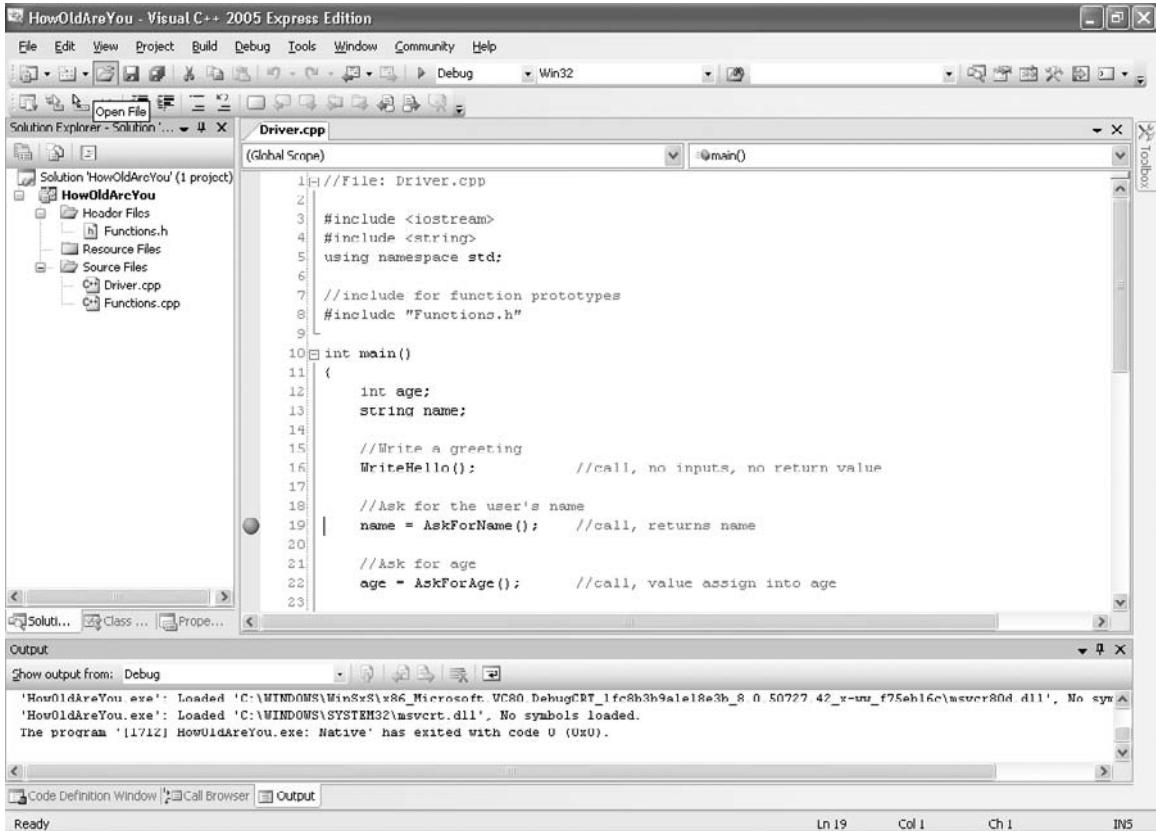
The program has executed through line 32 and is now stopped at line 33. Notice how the sphVol, barVol, and totalVol values are now seen in the lower left window.

The step into and step over features of the Visual C++ debugger works in the same manner for class functions as it does for standalone functions.

Debugger Features

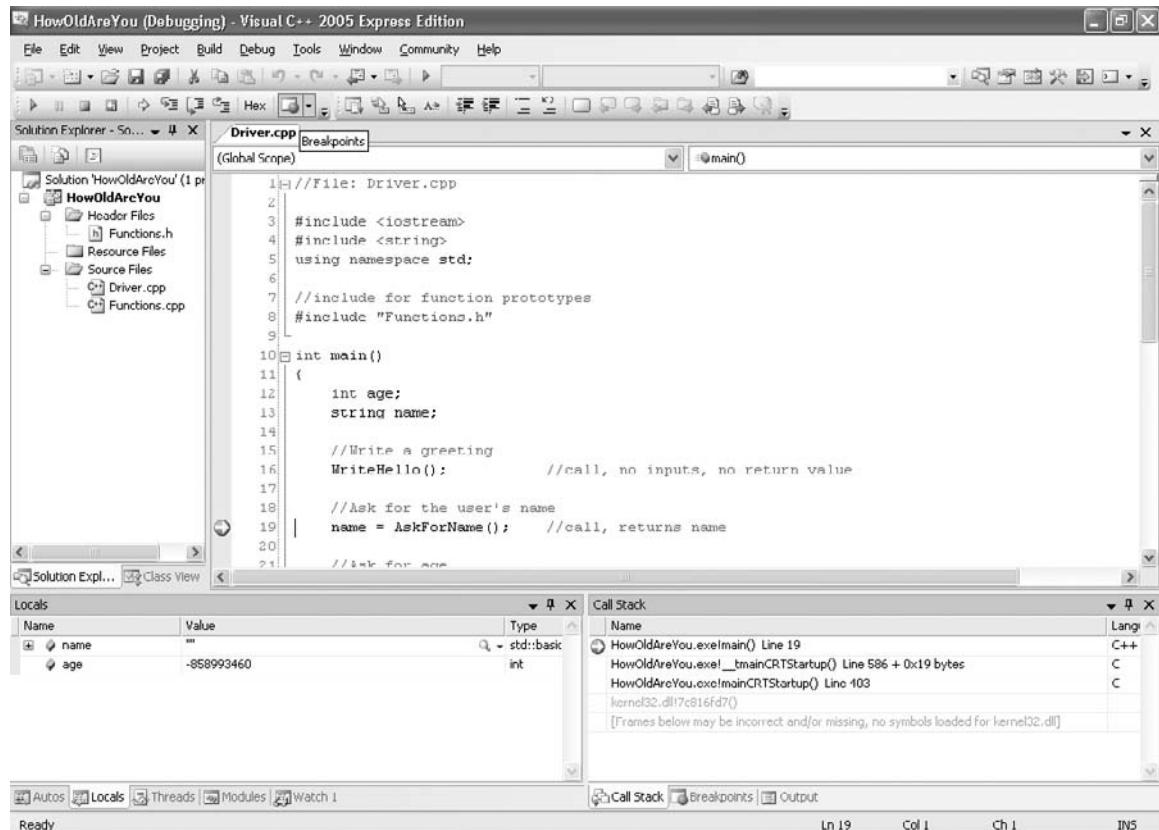
The Visual C++ 2005 Express debugger has several convenient features for the developer:

1. If you place the cursor over a variable in the debugger, a little window pops up and tells you the current value of that variable. This same feature will show you what data type the variable is when you are in the edit (non-debugger) mode. You can always use this cursor trick to see what the functions are for all the Visual C++ icons, including all the debugging icons.

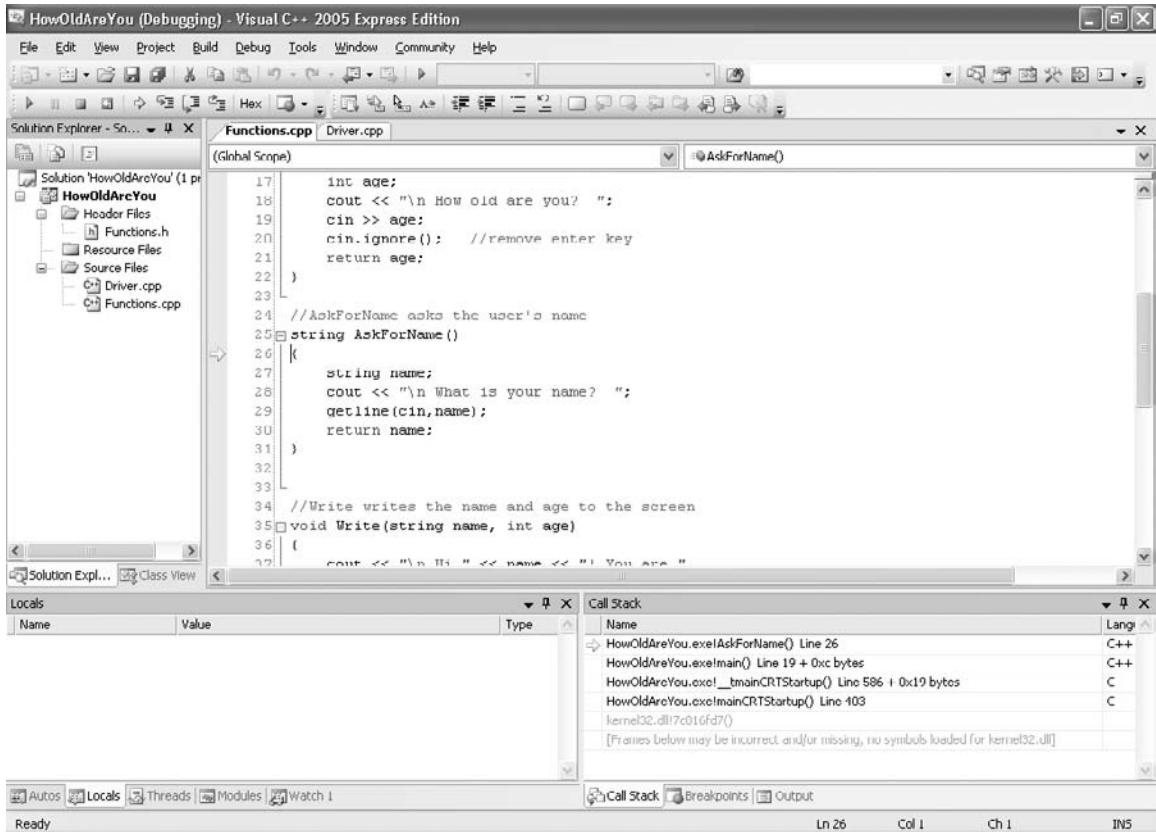
**Figure I-5**

Breakpoint at line 19 in the How Old Are You? program.

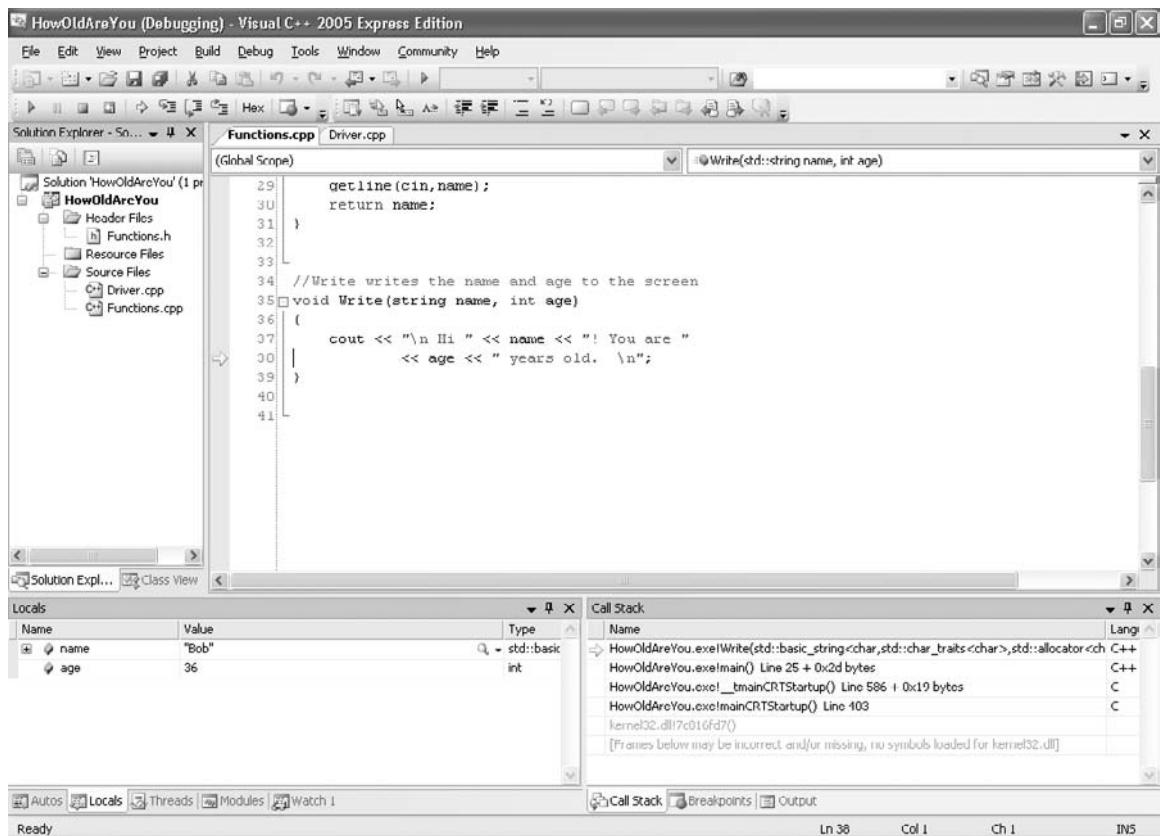
2. It is possible to set several breakpoints in your code and subsequently use the F5 key to run from breakpoint to breakpoint. Set the breakpoints, then start the debugger (hit F5). You may just hit the F5 key again to run to the next breakpoint.
3. The other handy debug window is the Call Stack window. It lists the order of the calls for the various functions. If a function calls a function that calls another function, it is easy to lose track of the order of the calls. By examining the Call Stack, you can see the line of function calls and who has called whom.

**Figure I-6**

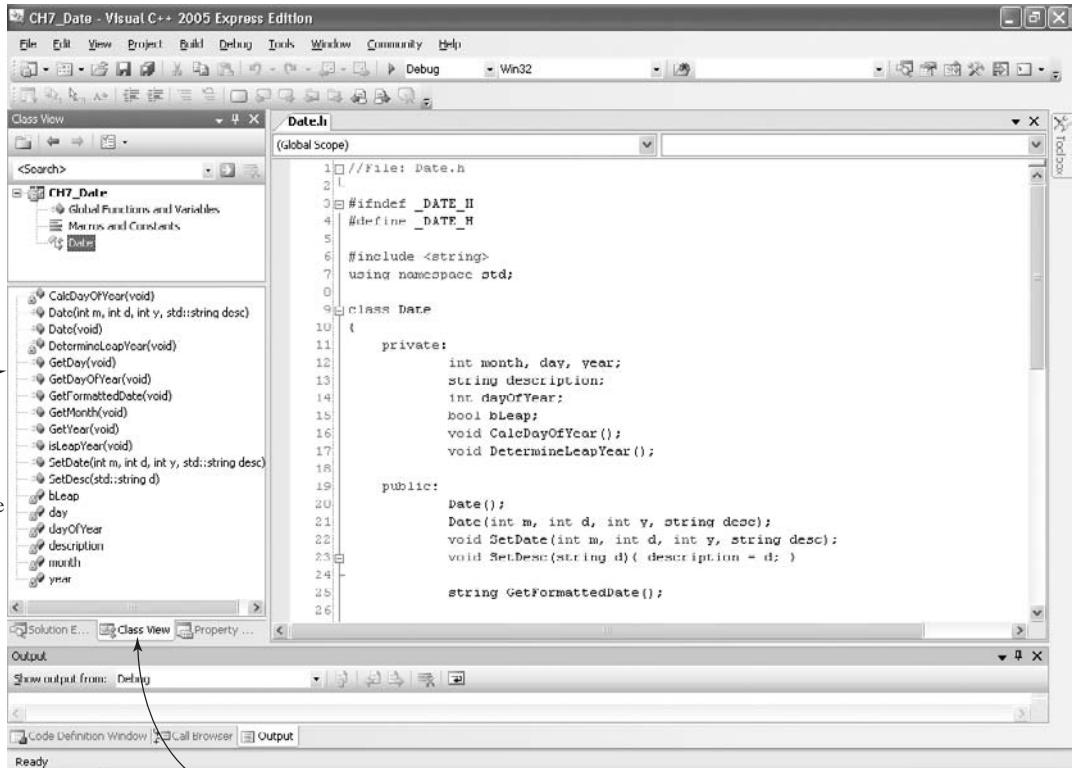
Debugger has stopped at the *AskForName()* function call.

**Figure I-7**

We have stepped into our *AskForName()* function.

**Figure I-8**

We have stepped into the *Write* function and can see our data variables.

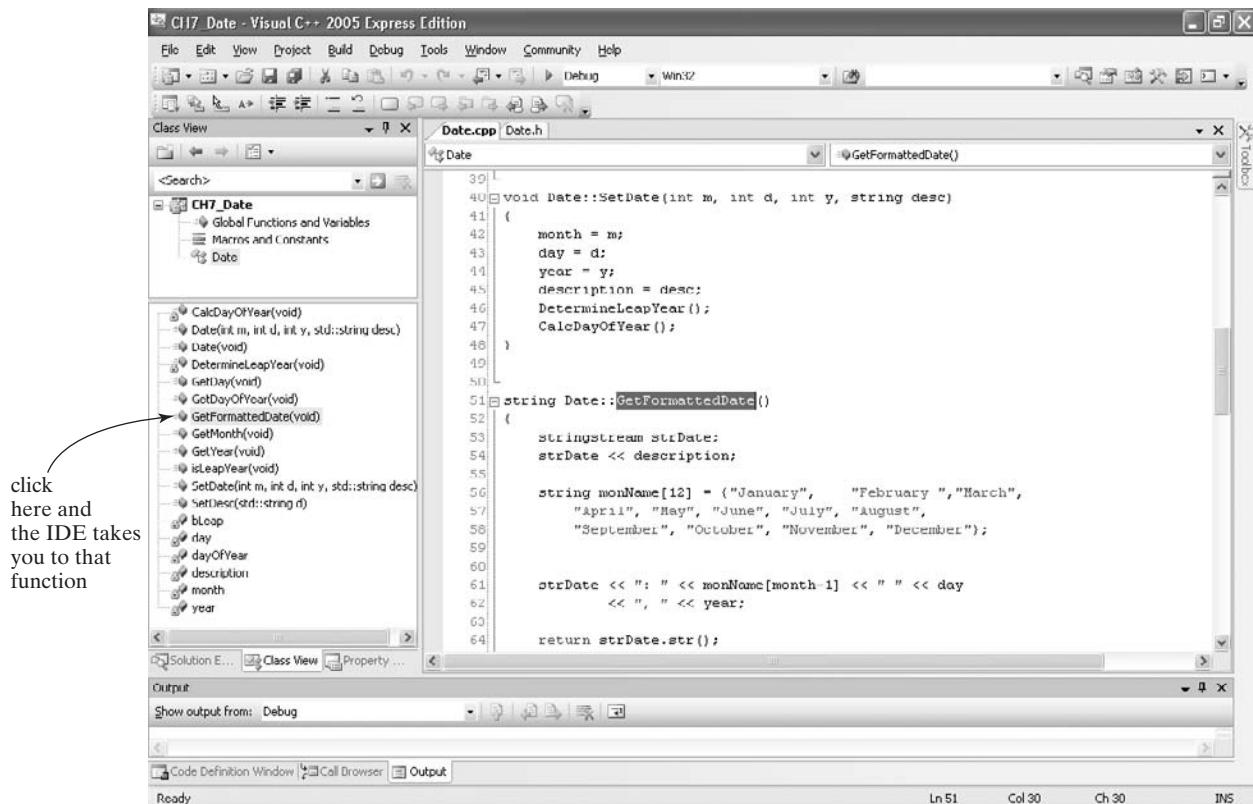


date,
class,
numbers
are seen here

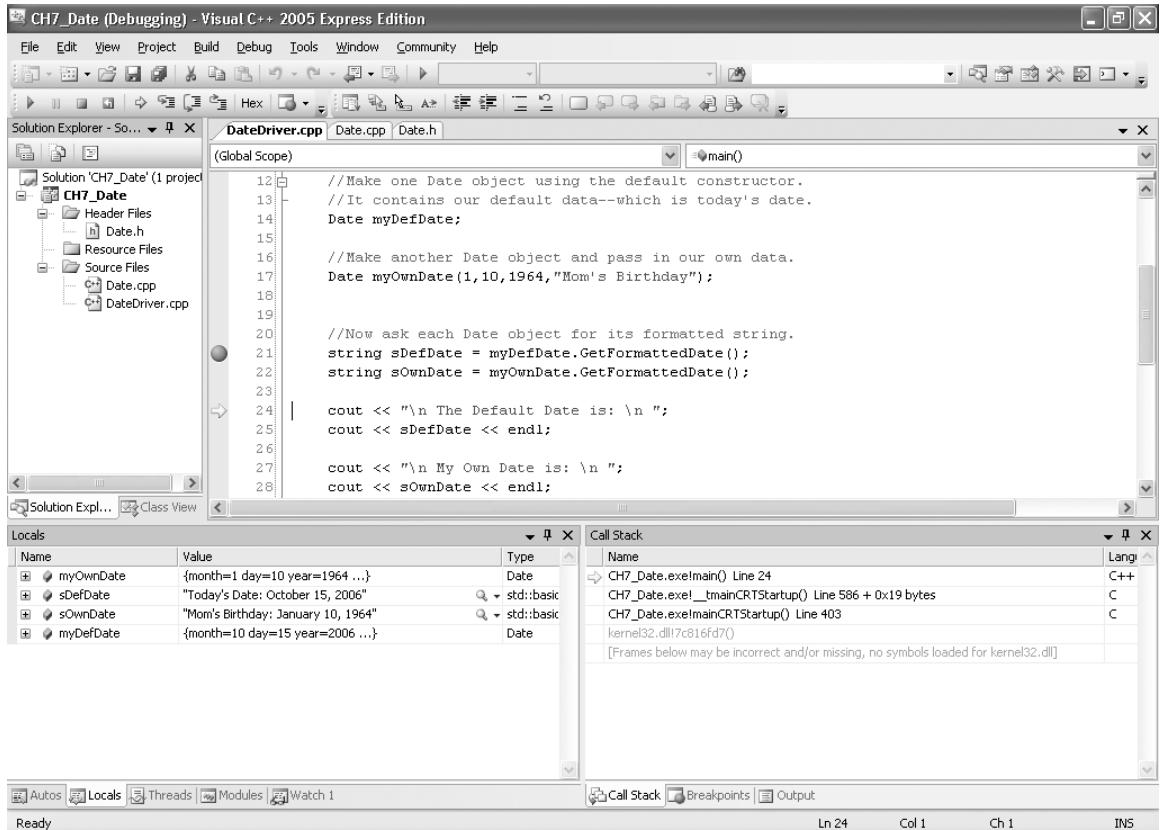
We are using the Class View tab to examine Date members

Figure I-9

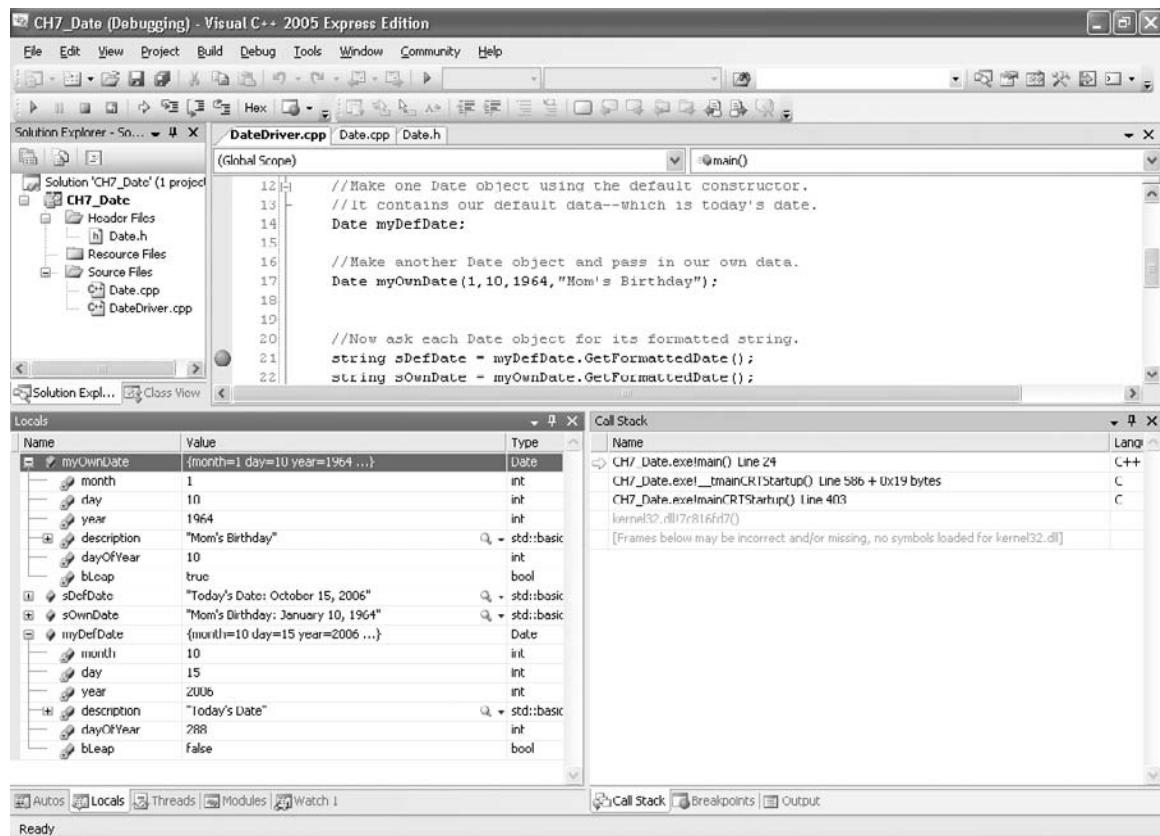
The Date class is seen in the Class View tab.

**Figure I-10**

Click on the class function name and Visual C++ takes you to the start of that function.

**Figure I-11**

The object data are seen easily in the debugger.

**Figure I-12**

Expanding the object variables shows all of the objects' data values.

This page intentionally left blank



Glossary

access specifier Dictates the accessibility of class members to the world. Access specifiers are private, protected, and public.

address operator: & When used with a variable name, the address operator returns the hex address of the memory location for that variable, for example:
`&number;`

algorithm A process or set of rules for solving a problem.

ANSI/ISO Standard American National Standards Institute/International Standards Organization. Oversees the committee that standardizes the C++ language.

arguments Input or return values for a function.

array A list of variables of the same data type that are referenced with a single name. Each element or member of an array is accessed by using an integer index value. The `[]` operators are used in the declaration statement. For example, an array of 100 doubles named “numbers” is declared by using:
`double number[100];`

array dimension The size of the array. Arrays can be single-dimensional (list), two-dimensional, (table), or multi-dimensioned.

array index The integer value that references a specific element or member of an array.

array pointers When an array is declared, C++ automatically creates a pointer of the same name that “points to” the first element of the array.

ASCII American Standard Code for Information Interchange (ASCII). The personal computer uses the ASCII notation for interpreting bits.

associativity When two (or more) operators that have the same procedure are found in a C++ expression, associativity specifies the order of operations, such as left to right or right to left.

automatic variable Variables that are declared inside functions. Automatic variables are also known as local variables.

base access specifier The three access specifiers, private, protected, and public, can also be used in the first line of a class declaration. When used here, the specifiers dictate how the base class members are treated in the derived classes.

base class A general-purpose class—also known as the “parent” class. New classes are derived from the base class. These new classes (children classes) are special cases of the base class. This principle is known as inheritance. For example, a vehicle class (which describes data and functions for motor vehicles) can be a base class. A Recreational Vehicle class can be derived from the Vehicle class. An RV is a Vehicle. The RV inherits properties from the Vehicle class.

base 16 Uses sixteen different symbols to represent a quantity of items. The sixteen symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F; this is also known as hexadecimal notation.

base 10 Our common counting system, with ten different symbols to represent a quantity of items. The ten symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; this is also known as decimal notation.

base 2 Uses two different symbols to represent a quantity of items. The two (most common) symbols are 0 and 1; This is also known as binary notation.

binary operator An operator that requires two operands, such as the addition operator $a + b$ or greater than operator $a > b$

bit A unit of data that exists in one of two states, such as 1 or 0, on or off.

block scope Variables declared within a set of {} are in scope as long as the program execution is within the {}.

body of function The statements of C++ code inside a function.

bool A data type used to declare boolean variables. A boolean variable contains the value true or false.

breakpoint A line of source code that is identified by the programmer while he or she is using the debugger. A breakpoint tells the debugger to stop the program on that line of code to enable the programmer to examine program variables.

byte Consists of 8 bits and is the basic unit of computer memory.

bytecode The Java compiler produces bytecode. The Java Virtual Machine program interprets the bytecode and issues machine instructions.

call-by-reference A term used when the address of a variable is passed to a function.

call-by-reference with pointers A term used when the address of a variable is passed into a pointer variable in the function. The address operator is required in the call statement and the indirection operator is required in the function.

call-by-reference with reference parameters The address of a variable is passed to a reference variable in the function. Just the name of the variable is required in the call statement and no indirection operator is required in the function.

call-by-value The value of a variable is passed to a function.

called function When one function uses (or invokes) a second function, the second function is the called function.

calling function When one function uses (or invokes) a second function, the first function is the calling function.

call statement The line of code in the program where a function is invoked.

case sensitive Recognizing upper- and lowercase letters to be different. C++ is a case-sensitive language. TOTAL, Total, and total are three different names in C++.

casting An operation in which the value of one type of data is transformed into another type of data.

C++ class The fundamental unit in object-oriented programming; contains the “job description” (member data and functions) for a program object.

char A data type used to declare a variable that contains one character symbol. The value ‘y’ can be stored in a char variable.

character string A character array, such as `char name[50];`

child class A derived class. *See* derived class *and* base class.

class A “job description” for a program object. The class contains member data and functions. *See* C++ class.

class constructor A function that has the same name as the class and is automatically called when objects are declared. The main job of a constructor function is to initialize object member data. It may be overloaded.

class declaration The source code that contains the class members, including the private, protected, and public sections with data and functions. Typically, the class declaration is contained in a *.h file.

class definition The source code that contains the implementation of member functions. Typically, the class definition is contained in a *.cpp file.

class members Either data or functions that are declared within a class.

class relationships The interactions and connections or associations that classes have with each other. There are three class relationships: uses a, has a, and is a.

comment Lines in source code, ignored by the compiler, in which programmers may write information concerning the file or program.

commercial off-the-shelf (COTS) software The general term for products offered by software developers. COTS is ready to use in the developer’s own software;—for example, COTS software can be file translators, image processing, or graphics software.

compiler A software program that reads source code and produces object or machine code.

compile-time polymorphism Illustrated in either overloaded functions or overloaded operators. (*See* polymorphism for complete discussion.)

complex class A class provided in the C++ language for working with complex numbers. (Complex numbers are represented as $a + bi$, where i is the $\sqrt{-1}$.)

conditional statement A statement in C++ in which a condition is evaluated. The result determines which path the program control takes.

const A modifier specifying that the variable is to remain a constant value, such as `const double pi = 3.14159265;`

constructor function See class constructor.

data structure A grouping of data variables that represents a logical unit, such as the data describing an item in a grocery store, including name, brand, price, UPC code, etc. The keyword *struct* is required.

data type A type of “container” holding program data. There are several different data types in C++, including *int*, *float*, *double*, *char*, and *bool*.

debugger The portion of the development environment that allows the programmer to step into a program and run the program line by line. The programmer can set breakpoints and watch how the values of variables change. It is possible to watch the order of statement execution in the debugger.

default input parameter list function Default input values may be supplied in the function prototype. If the variables (or values) are not provided in the function call, the default input values are used in the function.

delimiter A character specified in a read statement; the reading is concluded when the delimiter character is found.

derived class A new class created by inheriting data and functions from a base or parent class. See base class.

destructor function A function that has the same name as the class with a ~character, such as ~Date or ~Player. The destructor is called automatically when the object goes out of scope. It cannot be overloaded.

double A data type used to declare variables with up to ten places of decimal precision. The value 3.9342431337 can be stored in a double variable.

dynamic memory allocation Occurs when the program obtains memory for program variables “on the fly” as the program executes.

EBCDIC Extended Binary Coded Decimal Interchange Code. Mainframes and some minicomputers use the EBCDIC format for interpreting bits.

elements Members of an array.

enumerated data type A user-defined type; a numbered list of categories or items, such as days of the week, types of coins, or dog breeds.

error code A set of values specified by the programmer that represents errors discovered in the program. For example, error codes could be 1 for invalid user input, 2 for file not found, etc.

error flag The variable containing the error code.

error trap Term used by programmers that means “be on the look-out” for a possible error, such as an invalid user input or a divide by zero. Error trapping code checks and avoids invalid code operation.

exception handling A way to deal with run-time errors gracefully. When an error occurs, an exception is “thrown” and “caught” by an exception handler. An exception is another name for an error.

extract When characters are pulled out of the stream of data, and placed into variables, these characters are said to be extracted from the stream.

flags Variables used in programs to remember items or conditions in a program. Flags are usually integers or boolean variables. For example, set a flag = 0 if no errors occurred or flag = 1 if an error occurred.

float A data type used to declare variables with up to five places of decimal precision. The value 6.24313 can be stored in a floating point variable.

free The C standard library function that releases (frees) allocated memory from the program and returns it to the system.

function A discrete module or unit of code that performs specific tasks.

function body Consists of the statements of C++ code inside a function.

function header line The first line of any function in C/C++ contains the input types and arguments and return type.

function prototype The C++ statement that contains the function name, input, and return data types. The prototype informs the compiler that the function exists in the program.

GCC GNU C compiler A C++ development environment produced by the GNU project.

global variable A variable declared outside any function. All functions within the file can access global variables.

has a relationship The “has a” relationship is a class relationship in which one object (first) contains another object (second), and the second object is an integral part of the first object. If the second object were removed from the first object, the first object could not perform as expected. For example, a tennis player has a tennis racquet.

heap The portion of the computer memory in which global and static program variables are stored, along with the memory reserved during dynamic memory allocation (using the new operator).

hex address The address of a memory location.

hexadecimal notation The mathematical notation that describes how computer memory is referenced. Base 16 is hexadecimal notation.

identifier In C++, the name of a variable, label, function, or object that the programmer defines.

indirection operator * When used with a pointer, it directs the program to go to the address held in the pointer variable. The `->` is also an indirection operator used with a pointer to a struct or a class.

inheritance The inheritance process in an object-oriented language is demonstrated when a new class is created from an existing class—and this new class contains data and functions from the existing class. *See base class.*

inline functions Inline functions contain the inline keyword in the prototype. When the executable file is produced, the inline function body is embedded in the code at the function call location.

input arguments The variable values (or addresses) passed into a function.

instance of a class (or instantiation of a class) When a class variable is declared, an object is created. This object is known as an instance of a class.

int A data type used to declare variables that are whole numbers. The value of 42 can be contained in an integer variable.

is a relationship The “is a” relationship is found when one class is derived from a second class (inheritance-type class relationship). The second class becomes

a first class. For example, an Animal class can be a base class for a Bird class. The Bird object is a(n) Animal.

iterative and incremental development A process by which software is developed and tested in steps.

Java™ An object-oriented programming language originally developed by Sun Microsystems. Java programs can run on many different kinds of computers, consumer electronics, and other devices.

Java Virtual Machine A program that interprets Java bytecode and issues machine instructions.

Just in Time compilers Java compilers that build bytecode into code specific for the resident processor.

keywords In C++, words reserved by the language that have specific syntax and actions. They may not be used as variable or function names. For example, *switch*, *for*, *if*, and *float* are keywords in C++. (There are a total of sixty-three keywords in C++.)

linker A software program that hooks or links machine code and library code together to form an executable file.

local variable A variable declared either inside a function or in the function header line. Local variables exist only while the program control is in the function. The variables are lost when the function is exited.

logical operator An operator that evaluates AND, OR, or NOT conditions. The resultant value is either a 1 (true) or 0 (false).

loop A series of C++ statements enabling the program to repeat line(s) of code until a certain condition is met.

loop altering statement A line of C++ code that changes the value of a variable used in the condition-checking decision for a loop.

loop index A variable used as a counter in a loop.

lvalue A C++ entity that may be found on the left side of an assignment operator.

make file In the UNIX environment, a make file specifies the files and libraries for a program.

malloc The C standard library function that allocated memory as the program is executing.

memory The physical location in the computer that provides storage space for program data and variables.

method A class member function.

Microsoft Foundation Class (MFC) One implementation of an object-oriented environment for the development of application programs for Microsoft Windows. A programmer can use MFC to develop a Windows application without being an expert in Windows API. MFC has packaged, ready-to-go classes that provide Windows components as well as functions that handle the underlying window messages. See Windows API.

Microsoft Project Microsoft Visual C++ requires that the program files be contained in a project.

multiple-file programs Programs consisting of many source and header files that are created by the programmer.

multiple inheritance A term used to describe how two or more classes are used as base classes for a new class.

nested statements When one set of statements is located inside another set of statements, these statements are said to be nested.

newline '\n' or an Enter key.

null character A "zero" character; '\0'.

null-terminated string A character string has a null character that indicates the end of the pertinent data in the character array.

numeric classes C++ provides several classes for working with specialized numeric data, including the complex and valarray classes.

object A single instance of a class.

object model A software design technique that uses a collection of objects to represent the items in a program. Designing a program with the object model involves determining what objects are required and the relationships between them.

object-oriented program A program based on real-world items (objects) and how these objects interact with each other.

operand C++ operators work on operands. In the expression $a + b$, a and b are operands and $+$ is the operator.

operator function A class member function required to specify the task(s) for overloading an operator.

operators Symbols that direct certain operations to be performed, such as $+$ for addition, $=$ for assignment, etc.

out of bounds array When the program accesses array elements not legally declared, the program has gone out of bounds.

overloaded functions Functions that have the same name but different input parameter lists.

overloaded operator A valid C++ symbol (operator) that has been assigned a specialized task when the operator is used with objects.

parent class A base class. See base class.

pointer A variable that "points" to another variable. A pointer is a variable that "holds" another variable's hex address. When an indirection operator is used with a pointer, it directs the system to go where the pointer is pointing.

polling loop A loop in which the condition is based on continually checking the status of a variable until a certain condition is met. For example, a polling loop can be used to count off a certain number of seconds by getting the system time, adding the "waiting period" to the time, and then continually polling (asking) the system time until the waiting period has been reached.

polymorphism In an object-oriented language, allows the programmer to develop one interface and many implementations. There are two types of polymorphism in C++: compile time (overloaded operators and functions) and run-time (virtual functions).

portable language A language in which the source code does not need to be changed when it is moved from one type of computer to another.

precedence of operations A set of rules that dictates the order in which operations are performed.

preprocessor directives Lines in the source code that give the compiler instructions, such as `#include` or `#define` lines.

private specifier Used in class declarations as either an access specifier or as a base access specifier. The private access specifier dictates that only the member of the class can see and access the private members. Private members are not visible outside the class, and private members are not inherited in children classes. When private is used as a base access specifier, all public and protected members of the base class become private members of the derived class.

protected specifier Used in class declarations as either an access specifier or as a base access specifier. When used in a base class, the protected members are inherited in children classes. Protected members can be seen and accessed by all members of the class but are not visible outside the class. When protected is used as a base access specifier, all public and protected members of the base class become protected members of the derived class.

public specifier Used in class declarations as either an access specifier or as a base access specifier. When used in a base class, the public members are visible outside the class and are inherited in children classes. When public is used as a base access specifier, all public members of the base class become public members of the derived class. Protected members of the base class become protected members of the derived class.

pure virtual function Does not have an actual function implementation in the base class.

queue A class that is part of the C++ STL, a queue models standing in a line. It can contain data items, such as string, ints, etc.

qsort A general sorting routine located in the C standard library (stdlib.h).

reference parameter Declared by using the & operator. It is an implicit pointer.

relational operator An operator that evaluates `>`, `>=`, `<`, `<=`, `==`, `!=` conditions. The resultant value is either a 1 (true) or 0 (false).

return type The data type of the variable that is passed back to the calling function via a return statement.

rule of thirds A software development theory stating that software project time should be divided into three parts: designing, writing, and implementing/testing.

run-time polymorphism Illustrated in virtual functions. (*See polymorphism for complete discussion.*)

rvalue A C++ entity that may be found on the right side of an assignment operator.

sentinel value A trailer value.

sizeof In C++, returns the number of bytes reserved for the variable or for data type. For example, the statement:

`int bytes = sizeof(float);`

returns the value of 4 into the bytes variable.

software development skill set A list of the necessary skills a person should possess to become a successful programmer.

software development steps A series of logical steps followed to design and build software.

source code The file(s) that contain the C/C++ statements that provide program instructions.

stack The portion of the computer memory where local program variables are stored.

standardized libraries Libraries found in C++ that provide useful functions for mathematics, file I/O, text operations, etc. The cmath and iostream are standard library.

Standard Template Library (STL) A specific set of C++ libraries that contain many pre-written classes for the program to use. The STL includes vector and the queue classes used in this text.

static variable A local variable that retains its value until the program is terminated.

storage specifier A data type.

string class A class provided in the C++ language for working with text-based information (strings).

stringstream class A stream class that provides the program a way to place data (text and numeric) into an object and object a string from it. Contained in the C++ sstream library.

structure array Structures may be declared as arrays in the same manner that standard C++ data type arrays are declared.

structured programming A software design technique in which the program is based on the flow of the data through the program.

structure tag or template A blueprint for a structure variable. The structure tag is considered a data type in C++ and is used wherever data types are required (in variable declarations and functions).

syntax The grammatical rules required by the language. For example, after most C++ statements, the programmer must place a semicolon. If the semicolon is omitted, a syntax error is reported.

tag name A data type for variable declarations; used in the same manner as standard data types in function prototypes and function header lines.

templates Used to create generic functions and classes. The keyword, `template`, is used to set up the framework for a function or a class. In a function or class template description, the data type is actually passed in as an argument.

ternary operator An operator requiring three operands. The `? :` operators together are considered ternary operators because there are three operands. In the statement

```
int a, b;  
a = 50;
```

`b = a > 100 ? 200 : 0 ;`

the three operands are `a > 100`, `200`, and `0`.

top-down Top-down programming is a software design technique in which the program is based on the flow of the data through the program.

trailer value Used when reading data files. The trailer value is the last value in the data file and is different from the other data values in the file. The program can be structured to read until it sees the trailer value. For example, `a - 1` can be a trailer value for a file of positive integers.

true (1), false (0) Relational and logical operators return 1 if true and 0 if false.

unary operator An operator that requires only one operand, such as the increment operator, `++`.

user-defined data types Data structures, classes, and enumerations are all considered to be user-defined data types.

uses a relationship A relationship in which an object makes use of another object to accomplish a task. For example, a computer uses a printer.

using namespace std Directs the compiler to employ the standard (std) namespace, which includes the standard C++ libraries. To use the string class or the numeric classes provided in the C++ standard library, the headers must be declared in the new style format.

valarray class A class provided in the C++ standard libraries for working with arrays.

value The actual data stored in the variable is called the variable's value.

variable declaration The C++ program statement that dictates the type and name for a variable.

variables Memory locations reserved by the program for storing program data. The variable has a name and type, such as `int`, `float`, etc.

variable scope The length of time a variable is in existence as the program runs. Local variables in a function are in scope only when program control is in the function. Global and static variables are in scope until program termination.

vector A container that is used to hold data elements. The data is held in a linear list. It is a member of the C++'s Standard Template Library.

Visual C++ A development environment product from Microsoft.

watch variable A program variable that the programmer wishes to examine while the program is being run in the debugger. (The watch variable is being "watched" or examined.)

whitespace characters Spaces (blanks), tabs, line feeds, Enter key, control characters, vertical tabs, and form feed characters.

Windows Application Program Interface (Win32 API) The program interface that enables a programmer to develop Windows-based programs.

Windows messages Issued by the Windows operating system when certain events have occurred, such as a mouse click, keystroke, etc.

zero-indexed The first element of an array is referenced by using a zero [0] value. Such arrays are said to be zero-indexed.



Index

-- decrement operator, 65–66
- subtraction operator, 59
! NOT operator, 101–102
" " text start and end, 73
% modulus operator, 59,
 152–154
& address operator, 249–252,
 253–254, 260
&& AND operator, 101–102
* indirection asterisk operator,
 252, 412
* multiplication operator, 59
. dot operator, 80–82, 419
/ division operator, 59
// comments, 32, 41
? ternary operator, 117–118
// OR operator, 101–102
+ addition operator, 59
++ increment operator, 65–66
= assignment operator, 52
-> right arrow operator, 419
>> extraction operator, 50–52
#define statement, 67–69, 70,
 596

A

Abstraction, object-oriented
 principle of, 425
Access specifiers, 375–377
Access specifiers, 472, 481–483
base class, 481–482

multiple inheritance and,
 481–483
private members, 472
protected members, 472
public members, 472
Accumulation operators, 66–67
Addition (+) operator, 59
address (&) operator, 249–252,
 253–254, 260
data variables and, 249–252
pointers and, 253–254
 references and, 260
Algorithm design, 26–27
American National Standards
 Institute (ANSI), 6
AND (&&) operator, 101–102,
 144, 146–148
Arguments, functions as, 184
Arithmetic operators, 53, 59–65,
 412
<*cmath*> command, 65
addition (+), 59
 division (/), 59
 fractional calculations, 61–62,
 62–65
intermediate results with, 59–61
modulus (%), 59
multiplication (*), 59
operand, 59
overloaded, 412
precedence of, 53
subtraction (-), 59
temperature conversions, 62–65
Array of chars, 42
Arrays, 292–367, 407–412
call-by-reference, 306–308

character, 295, 312–322
Colorado river stream gauge
 flow data program exam-
 ple, 352–356
compiler errors, 341–342
C-strings, 295, 312–322
data files and, 333–340
data variables used in, 293–294
declaration, 298–300
defined, 293
element, 294
errors encountered with,
 341–342
filing from data files, 333–340
FindFavorite function
 program examples,
 342–346
for loops and, 296–298
functions and, 305–312,
 327–333
fundamentals of, 294–304
index, 294
initialization, 298–300
link errors, 342
multidimensional, 322–327,
 327–333
null-terminated character
 strings, 295
objects in classes, 407–412
out of bounds, 300–303
passing to functions, 306–308
phone list objects program
 example, 408–412
pointers, 305–306
random numbers, generating
 and sorting using, 308–312

seven dwarfs and files program
example, 346–351
single-dimensional, 294
variables, 294–295
vectors, comparison with,
303–304
zero indexed, 296
ASCII character codes, 555–561
Ask() function, 401–402
Assignment operator (=), 52
Associativity, defined, 53
at() function, 142–144
atof function, 317
atoi function, 317
atol function, 317
Automatic variables, 205

B

Base class, 469–470, 481–482,
483–484, 490, 491–500
access specifiers, 481–482
constructor functions,
483–484, 490
defined, 470
destructor functions, 484
inheritance and, 469–470
multiple inheritance and,
481–482
specifiers in derived class
declarations, 482
virtual functions in, 491–500
Binary file input/output, 577–581
Binary operators, 101–102,
412–418
classes, in, 412–418
conditional statements, as,
101–102
overloaded, 412–418
Bits, defined, 43
Block scope, 205
Boolean values, functions and,
217–221
Braces, 109, 123
if statements, using with, 109
loops, using with, 123
break; statements, 122, 134–135
Bytes, 43, 562–563
computer memory and, 562–563

defined, 43
format conventions, 563

C

C language, 6–10
compiled language, as a, 8–10
history of, 6–7
C++ software, 2–23, 24–99,
527–504, 541–554, 555–561
algorithm design, 26–27
American National Standards
Institute (ANSI), 6
ASCII character codes for,
555–561
compiled language, as a, 8–10
construction techniques,
19–20
cross-platform code libraries
and, 8
developer skill set, 3–4
development environment
tools, 527
Graphical User Interface GUI
framework, 8
Help system, 535–538
history of, 7
installation of, 528
International Standards
Organization (ISO), 6
introduction to, 3
keyword dictionary, 541–553
object-oriented programming
and, 11–13, 13–19
open source libraries, 10
operators in, 554
program project construction
steps, 528–535
programmers, rules for, 5–6
programming fundamentals,
25–29
source code, 5
steps to programming success,
28–29
structured programming, 11,
13–19
terminology for, 29–30
textbook concerns of, 4–5
top-down programming, 11

troubleshooting, 20–21
wxWidgets, 8
C-strings, 295, 312–322, 342–346,
346–351
atof function, 317
atoi function, 317
atol function, 317
char function, 319
defined, 295
FindFavorite function
program examples,
342–346
functions in *cstring* library,
320
initialization, 313
input, 315–319
null character, 312, 313–315
null-terminated character
strings, 295
seven dwarfs and files program
example, 346–351
Call statements, 186, 194–195
Call-by-value functions, 195–198
Call-by-reference function,
255–259, 260–262, 306–308
passing arrays to functions
using, 306–308
pointers, using, 255–259
references, using, 260–262,
306–308
Calling functions, 186, 194–195,
195–198, 255–259, 380–382
address (&) operator, 260
call statements, 186, 194–195
call-by-reference, 255–259,
260–262
call-by-value, 195–198
cout statement, 382
indirection asterisk (*)
operator, 254–259
writing classes, for, 380–382
char data type, 42
char function, 319
Character arrays, *see* C-strings
Child class, 465
cin object, 50–52, 78–79, 140–141
debugging and, 140–141
extraction operator (>>), 50–52

- reading data using, 50–52, 78–79
string class and, 79, 140–141
- Class constructor functions, 377–378
- Class definition, 374
- Class destructor function, 404–407
- Class member, 374, 399–404, 426, 427
- Ask()* function, 401–402
- defined, 374
- have you had your birthday program example, 399–404
- object-oriented principle of, 426
- objects as a, 399–404
- overloaded, programming error due to, 427
- Write()* function, 401–402
- Classes, 12, 78–82, 142–144, 211–213, 265–268, 334–340, 465, 469–470, 472, 481–482, 490, 491–500.
- See also* Inheritance; Writing classes
- access specifiers, 472, 481–483
- base, 469–470, 481–482, 483–484, 490, 491–500
- binary operators in, 412–418
- child, 465
- constructors, 377–378, 483–490
- defined, 12
- derived, 469–470, 481–482, 483–484, 490
- destructor function, 404–407, 483–490
- errors while writing, 424–427
- get* function, 391–395, 440–444
- ifstream*, 334–340
- object-oriented principles, 424, 425
- objects and, 78–82, 368–463
- ofstream*, 334–340
- overloaded operators, 412–418
- parent, 465
- pointers and, 419–424
- queue *<queue>*, 265–268, 370–371
- references and, 419–424
- relationship of in inheritance, 469
- set* function, 391–395, 440–444
- stock tank calculator program example, 440–446
- string *<string>*, 78–80, 370
- stringstream* *<sstream>*, 211–213
- switch* statement, 444–446
- teacher's helper, grader program example, 433–440
- unary operators in, 412–418
- vector* *<vector>*, 142–144, 370
- virtual functions in, 491–500
- writing, 371–399
- Comments *(//)*, 32, 41
- Commercial off-the-shelf (COTS) software, 500
- Compiler, defined, 8
- Compiler errors, 37–39, 214, 215–216, 271–272, 341–342
- arrays, 341–342
- cannot convert parameter, 271, 341–342
- function headers missing, 215–216
- functions and, 214, 215–216, 271–272
- illegal indirection, 271–272
- language syntax and, 37–39
- parameters unaccepted, 214
- writing functions, 214, 215–216
- Compile-time polymorphism, 491
- Computer memory, 247–249, 562–567
- byte formats, 562–563
- disks and, 562–567
- functions and, 247–249
- hex address, 247–248
- hexadecimal notation, 247, 563–566
- media, 563
- Concurrency, object-oriented principle of, 425
- Conditional statements, 101–103, 101–103, 136–137, 144, 146–148
- AND (*&&*) operator, 101–102, 144, 146–148
- logical operators, 101–103, 104
- NOT (*!*) operator, 101–102, 144, 146–148
- OR (*//*) operator, 101–102, 144, 146–148
- relational operators, 101–103, 104, 136–137
- const* access modifier, 69–70
- Constants, interpretation of for precedence of operations, 55–56
- Constructors, 377–378, 483–490, 582, 586
- base class, 483–484, 490
- class functions, 377–378
- derived class, 483, 490
- doctor is a person program example, 484–490
- inheritance and, 483–490
- overloaded, 377, 484
- passing parameters into overloaded, 484
- string class, 582
- vector* class, 586
- Containers, C++ programming, 42–43
- continue* statements, 134, 135
- Control statements, 100–177
- AND (*&&*) operator, 101–102, 144, 146–148
- braces, use of with, 109, 123, 138–139
- break ;*, 122, 134–135
- conditional statements, 101–103, 144, 146–148
- continue*, 134, 135
- counter values, 136
- debugging, 140–141
- else-if*, 110–112
- goto*, 134
- if*, 103–117, 139–140
- if-else*, 107–117
- infinite series calculation program example, 148–150
- jump statements, 134–135
- logical operators, 101–103, 104

- loops and, 122–134
NOT (!) operator, 101–102,
 144, 146–148
OR (/) operator, 101–102,
 144, 146–148
infinite series (PI) calculation
 program example, 148–150
pond pump calculator program
 example, 162–165
precedence of operators in, 103
program flags, 154–158
random number generator
 program example,
 150–152, 152–154
relational operators, 101–103,
 104, 136–137
return, 134
search for the name program
 example, 154–158
semi-colon placement,
 138–139
switch, 118–122
ternary (?) operator, 117–118
time conversion program
 example, 158–160
troubleshooting, 136–141
variable evaluations, 137–138
vector class, 142–144
vowel counting program
 example, 160–162
Counter values, 136
cout object, 34–35, 73–74, 74–76,
 78, 382
 calling class functions using,
 382
 escape sequences and, 73–74
ios formatting flags and,
 74–76, 84–85
 screen output and, 34–35
 writing data using, 78
Cross-platform code libraries, 8
- D**
- Data casts**, 70–73
Data files, 333–340, 352–356,
 568–581
 arrays and, 333–340
 binary input/output, 577–581
- Colorado river stream gauge**
 flow data program
 example, 352–356
format of, 568–570
header line, 568, 569
ifstream class, 334–340
input/output, 568–581
iostream library, 570–577
maximum and minimum arrays,
 finding with, 337–340
ofstream class, 334–340
phone bill program example,
 334–337, 368–340
reading to end of, 569–570
sentinel value, 568–569
trailer value, 568–569
- Data types**, 41–46, 53–55
 array of chars, 42
 bits, 43
 byte, 43
char, 42
 containers, 42–43
 defined, 42
double, 42
float, 42
int, 42, 45
 integer size, 45–50
 labels, 42–43
 modifiers, 43–45
 precedence of operations,
 53–55
 stored values of, 53–55
string, 42
 values, 42–43
 variables, 42–43
- Data variables**, 245–249,
 249–250, 269, 293–294
 address (&) operator, 249–252
 arrays, using single in, 293–294
 memory and, 245–249
 properties of, 250–251, 269
sizeof operator, 246–247
- Debugging**, 140–141, 600–613
 C++ features, 605–613
cin object, 140–141
classes, 604–605
 concepts of, 600
 functions, 603–604
- getline() function**, 140–141
string class, using, 140–141
terminology for, 601
- Declaration**, 46–48, 186–187,
 191–193, 198–200,
 298–300, 323, 374
- arrays**, 298–300, 323
- class**, 374
- function header line**, 47–48, 186
- functions**, 186–187, 191–193
 multidimensional arrays, 323
 prototype statements, 191–193
 undeclared identifiers, 198–200
variables, 46–48
- Decrement operators (--)**, 65–66
- Default input parameter list**
 functions, 202–204
- Definition**, functions, 186–187,
 192–200
- Derived class**, 469–470, 481–482,
 483–484, 490
- base class specifiers in**
 declarations, 482
- constructor functions**,
 483–484, 490
- defined**, 470
- destructor functions**, 484
- inheritance and**, 469–470
- multiple inheritance and**,
 481–482
- overloaded constructor**
 functions, 484
- Destructors**, 404–407, 483–490
 class functions, 404–407
 inheritance and, 483–490
 overloaded, 377, 484
- Division (/) operator**, 59
- do while loops**, 132–134
- Dot operator (.)**, 80–82
- double data type**, 42
- Dynamic memory allocation**,
 264–265
- E**
- else-if statements**, 110–112
- empty() function**, 266
- Encapsulation**, object-oriented
 principle of, 424, 425

Errors, 37–39, 214–216, 271–272, 341–342, 424–427. *See also Compiler errors; Link errors*
 compiler, 37–39, 214, 215–216, 271–272, 341–342
 language syntax and, 37–39
 link, 214–215, 342
 overloaded member functions, 427
 undeclared identifier, 424–426
 unresolved external, 214–215, 342, 426–427
 writing classes, 424–427
 writing functions, 214, 215–216
 Escape sequences, 34, 73–74
 C++ use of, 73
 new line output (*\n*), 34, 73–74
 types of, 77
extern keyword, 598–599

F

Filing arrays from data files, 333–340
find function, 81
float (floating point variable) statements, 137–138
float data type, 42
for loops, 124–128, 296–298, 324 arrays and, 296–298, 324 control statements, 124–128 nested, 324 two-dimensional arrays, 324
 Functions, 11, 31, 33–34, 178–243, 244–291, 305–312, 327–333. *See also Virtual functions*
#include statement, 187
 address (&) operator, 249–252, 253–254, 260
 arguments, 184 arrays and, 305–312, 327–333 basic format of, 183–186 bathtub volume calculator program example, 282–285
 body, 186, 192–194
 Boolean values, 217–221 calculating pay program example, 223–227

call-by value, 195–198
 call-by-reference, 255–259, 260–262, 306–308
 call statements, 186, 194–195
 called, 186
 calling, 186, 194–195, 195–198, 255–259
 compiler error, 214, 215–216, 271–272
 data variables, 245–249, 249–250
 declarations, 186–187, 191–193
 default input parameter list, 202–204
 defined, 11, 31, 179
 definition, 186–187, 192–194
 errors encountered with, 213–216, 270–272
function_name, 183–184
 header line, 33–34, 192–194, 215–216
 I scream for ice cream program example, 275–277
 infinite series (PI) program example, 148–221–223
 input parameters, 184, 214
 Is it prime? program example, 216–221, 272
main, 31, 33, 180, 187
 memory, 245–249
 multiple files program examples, 227–231, 231–235
 overloaded, 200–202, 223–227
 pointers, 252–254, 254–260, 264–265, 269, 270–272, 275–277, 305–306
 program flags, 216–217
 prototype, 186–187, 191–192
 queue class *<queue>*, 265–268, 280–282
 random numbers, generating and sorting using, 308–312
 references, 260–264, 269, 270–271, 277–280, 306–308
 requirements for writing, 186–200
return statement, 193–194

returning items from, 244–291
 search for woman's name program example, 277–280
sizeof operator, 246–247
 song organizer program example, 280–282
stringstream class *<sstream>*, 211–213
 variables in, 204–211, 250–251, 269
 writing, 179–243

G

get function, 391–395, 440–444
getline() function, 79, 140–141
 debugging and, 140–141
 reading strings using, 79
 Global variables, 205–209
goto statements, 134
 Graphical User Interface (GUI) framework, 8

H

Header file, 378–379
 Header line, 33–34, 192–194, 568–569
main function, 33–34
 data files and, 568–569
 function, 33–34, 192–194
 Hello World!, program example, 31–34
 Help system, 535–538
 Hex address, 247–248
 Hexadecimal notation, 247, 563–566
 How's the Weather, program example, 34–35

I

Identifier naming rules, 47
if-else statements, 107–117
if statements, 103–117, 139–140, 145
 braces, using with, 109, 139–140
else-if statements, 110–112, 145

- examples of, 103–107
if-else statements, 107–117,
 145
inefficient programming with,
 112–113
nested *if-else*, 114–117
this old man program example,
 113–114
troubleshooting mistakes,
 139–140
use of, 145
ifstream class, 334–340
Increment operators (++), 65–66
Indirection asterisk (*) operator,
 252, 254–260, 412
Infinite loop, 123–124
Infinite series calculation program
 example, 148–150
Inheritance, 425, 464–526
 access specifiers, 472, 481–483
 base class, 469–470, 481–482,
 483–484, 490, 491–500
 basics of, 469–480
 child class, 465
 class relationship, 469
 commercial off-the-shelf
 (COTS) software, 500
 constructors, 377–378, 483–490
 counter class program exam-
 ples, 467–469, 470–472
 date class program example,
 506–510
 defined, 469
 derived class, 469–470,
 481–482, 483–484, 490
 destructors, 483–490
 employees, bosses, and CEOs
 program example, 475–480
 ice cream dialog program
 example, 467
 multiple, 481–483
 object-oriented principle of,
 425
 parent class, 465
 pay calculations program
 example, 500–506
 polymorphism and, 490–500
 protected members, 472–475
rain barrel estimator program
 example, 510–518
 virtual functions and, 491–500,
 500–506
Input/output, *see* Data files
Input parameters, functions, 184
Installation of C++, 528
int (integer value) statements,
 137–138
int data type, 42, 45
Integer size, C++ programming,
 45–50
International Standards
 Organization (ISO), 6
Invoking object, 417–418
ios formatting flags, 74–76, 84–85
iostream library, 570–577
Iterative and incremental
 development, 28–29
- J**
- Jump statements, 134–135
- K**
- Keyboard input, 50–52, 73–78,
 78–82
 cin object, 50–52, 78–79
 extraction operator (>>),
 50–52
 obtaining data from, 50–52
 screen output and, 73–78
 string class and, 78–82
 text start and end (" "), 73
Keyword dictionary, 541–553
Keywords, 39, 40
- L**
- Labels, C++ programming, 42–43
Link error, 214–215, 342
 arrays, 342
 functions, 214–215
 unresolved external, 214–215,
 342
Local variables, 205
Logical operators, 101–103, 104,
 412
 conditional statements, as,
 101–103, 104
- defined, 101
overloaded, 412
long data type modifiers, 43, 45
Loops, 122–134, 139–140, 146,
 296–298, 324
 braces, using with, 123,
 139–140
 control statements and,
 122–134
 defined, 122
 do while, 132–134
 for, 124–128, 296–298, 324
 index, alteration of, 126
 infinite, 123–124
 troubleshooting mistakes,
 139–140
 use of, 146
 while, 122–123, 128–132
- M**
- Main function for writing classes,
 379
main function, 31, 33, 180, 187
Make file utility package, 29
Member, *see* Class member;
 Protected members
Memory, 245–249
 computer, 247–249
 data variables and, 245–249
 dynamic allocation, 264–264
 hex address, 247–248
 hexadecimal notation, 247
 random access (RAM), 245
 reserving, 247
Modifiers, C++ programming,
 43–45
Modularity, object-oriented
 principle of, 425
Modulus operator (%), 59,
 152–154
Multidimensional arrays,
 322–327, 327–333, 346–351
bingo program examples of
 two-dimensional, 324–327,
 328–332
declarations, 323
functions and, 327–333
initialization, 323–324

- nested *for* loops in, 324
 seven dwarfs and files
 program example, 346–351
- Snow White example of
 names program, 332–333
- two-dimensional, 323–327,
 327–333
- Multifile programs, 588–600
- Multiple inheritance, 481–483
- Multiplication (*) operator, 59
- N**
- Naming rules, 47
- Nested *if-else* statements,
 114–117
- NOT (!) operator, 101–102, 144,
 146–148
- Null character, 312, 313–315
- Null-terminated character
 strings, 295
- O**
- Object-oriented analysis
 (OOA), 426
- Object-oriented design (OOD),
 426
- Object-oriented programming,
 11–13, 13–19, 424–425. *See also* Classes; Objects
- ATM example using, 13–16
 benefits of, 12–13
 C++ class, 12
 car maintenance example
 using, 16–19
 defined, 11
 encapsulation, 424, 425
 language principles, 424, 425
 object, 12
 structured programming
 versus, 13–19
- Objects, 12, 78–82, 368–463
cin, 78–79
 arrays of, 407–412
 big, bigger, biggest football
 players program example,
 427–433
- class destructor function,
 404–407
- classes and, 78–82, 368–463
cout, 78, 382
 defined, 12
 dot operator (.), 80–82
 Have you had your birthday?
 Program example, 399–404
- invoking, 417–418
 overloaded operators and,
 412–418
 pointers and, 419–424
 references and, 419–424
 string class and, 78–82
- ofstream* class, 334–340
- Open source C++ libraries, 10
- Operands, 101
- operator* function, 412
- Operators, 10, 48–67, 101–103,
 117–118, 412–418, 554.
See also Precedence of
 operations
- accumulation, 66–67
 arithmetic, 53, 59–65
 assignment (=), 52
 associativity, 53
 binary, 101–102, 412–418
 C++ software, 554
cin object, 50–52
 conditional statements and,
 101–103, 117–118
 decrement(--), 65–66
 defined, 10, 48
 extraction (>>), 50–52
 increment (++), 65–66
 keyboard, obtaining data
 from, 50–52
 keyword, 412
 logical, 101–103
 objects and, 412–418
 operands, 101
 overloaded, 412–418
 precedence of, 52–58, 103
 relational, 101–103
 road trip calculation program
 example, 48–50
 ternary (?), 117–118
 unary, 101–102, 412–418
- OR (//) operator, 101–102, 144,
 146–148
- Out of bounds arrays, 300–303
- Overloading, 200–202, 223–227,
 377, 412–418, 427–433
- big, bigger, biggest football
 players program example,
 427–433
- calculate pay program example,
 223–227
- class constructor function, 377
- functions, 200–202, 223–227
- indirection asterisk (*)
 operator, 412
- member function, errors
 writing classes using, 427
- objects, 412–418
- operators, 412–418, 427–433
- P**
- Parent class, 465
- Persistence, object-oriented
 principle of, 425
- Pointers, 10, 252–254, 254–260,
 264–265, 269, 270–272,
 275–277, 305–306, 419–424
- address (&) operator, 253–254
- array, 305–306
- call-by-reference function,
 255–259
- compiler errors, 271–272
- data items, efficient handling
 of large, 259–260
- date time demo program
 example, 244–424
- defined, 10, 252
- dynamic memory allocation,
 264–265
- errors encountered with,
 270–272
- I scream for ice cream program
 example, 275–277
- illegal indirection, 271–272
- importance of, 264–265
- indirection asterisk (*)
 operator, 252, 254–260
- objects and, 419–424
- reference parameters, 260, 262
- references and, 262–264, 269
- right arrow operator (->), 419

- Polymorphism, 425, 490–500
 compile-time, 491
 defined, 490
 object-oriented principle of,
 425
 run-time, 491
 virtual functions and, 490–500
- pop()* function, 265–266
- Portable language, defined, 8
- Precedence of operations, 52–58,
 103
 arithmetic operators and, 53
 associativity, 53
 conditional statements and,
 103
 constants, interpretation of
 for, 55–56
 data types, stored values of,
 53–55
 evaluating expressions and,
 103
lvalue object, 57–58
rvalue object, 57–58
 variables, 56–57
- Preprocessor directives, 32–33,
 378, 594
- Private access specifiers, 375
- Program flags, 154–158, 216–217
 control statements and,
 154–158
 functions and, 216–217
- Program project construction
 steps, 528–535
- Programming fundamentals,
 24–99, 140–141. *See also*
 Classes; Functions; Objects
- algorithm design, 26–27
 calculating the volume of a
 dumbbell program
 example, 82–84
 case sensitivity, 31
 character data, 84–85
 character data program
 example, 84–85
 classes, using, 78–82
 comments (*//*), 32, 41
 compiler errors, 37–39
const access modifier, 69–70
- data and data types, 41–46, 53
 data casts, 70–73
 debugging, 140–141
#define statement, 67–69, 70
 escape sequences, 73–74
 functions, 31, 33–34
 good style for, 39–41
 Hello World! program example,
 31–34
- How's the Weather program
 example, 34–35
- #include* statement, 32–33
- I scream, you scream, we all
 scream for ice cream pro-
 gram example, 86–87
- ios* formatting flags, 74–76,
 84–85
- keyboard input, 50–52, 73–78,
 78–82
- keywords, 39, 40
- language syntax, 37–39
- love my style program example,
 40–41
- main* function, 31, 33
- objects, using, 78–82
- operators, 48–67
 practice programs, 82–90
 preprocessor directives, 32–33
 project, 29
 screen output, 34–35, 73–78,
 78–82
- statements, 34
 steps to programming success,
 28–29
- stream IO manipulators, 76–78
- string class, using, 78–82
 terminology for, 29–30
 variable declaration, 46–48,
 56–58
- water facts and stock tanks for
 livestock program example,
 88–90
- Project, defined, 29
- Protected members, 472–475
- Prototype, functions, 186–178,
 191–192
- Public access specifiers, 375
- Pure virtual function, 492
- push()* function, 265–266
push_back() function, 142–144
- Q**
- Queue class *<queue>*, 265–268,
 280–282, 370, 586–587
 functions, 586–587
 song organizer program
 example, 280–282
 use of, 265–268, 370
- R**
- rand()* function, 150–152
- Random access memory
 (RAM), 245
- Random numbers, 150–152,
 152–154, 308–312
 arrays, generating and sorting
 using, 308–312
 bubble sort, 309–312
 functions, generating and
 sorting using, 308–312
 generator program example,
 150–152, 152–154
 modulus (%) operator,
 152–154
rand() function, 150–152
srand() function, 150–152
- Reading to end of files, 569–570
- Reference parameters, 260, 262
- References, 10, 260–264, 269,
 270–271, 277–280,
 306–308, 419–424
- address (&) operator, 260
- call-by-reference, 260–262,
 306–308
- compiler errors, 271–271
- date time demo program
 example, 244–244
- defined, 10
- dot operator (.), 419
- errors encountered with,
 270–271
- objects and, 419–424
- parameters, 260, 262, 269
- passing arrays to functions,
 306–308
- pointers and, 262–264

search for woman's name
 program example, 277–280

Relational operators, 101–103,
 104, 136–137, 412
conditional statements, as,
 101–103, 104
defined, 101
errors programming with,
 136–137
overloaded, 412
return statement, 134, 193–194
Run-time polymorphism, 491

Scope operator, 384–375
Screen output, 34–35, 73–78,
 78–82
 cout objects, 34–35, 74–76, 78
 escape sequences, 34, 73–74
 ios formatting flags, 74–76,
 84–85
 keyboard input and, 73–78
 new line (/n), 34, 73–74
 set(w) manipulator, 76
 stream IO manipulators, 76–78
 string class and, 78–82
 text start and end (" "), 73
Semi-colon placement, 138–139
Sentinel value, 568–569
set function, 391–395, 440–444
setw() manipulator, 76–78
short data type modifiers, 43, 45
size() function, 142–144, 266
sizeof operator, 246–247
Source code, 5, 8
srand() function, 150–152
Standard Template Library, 11,
 142
Static variables, 209–211
Stream IO manipulators, 76–78
String class *<string>*, 78–80,
 140–141, 370, 582–585
 cin objects, 79, 140–141
 constructors, 582
 debugging using, 140–141
 dot operator (.), 80–82
 editing functions, 584
 find function, 81

getline() function, 79,
 140–141
miscellaneous functions, 585
overloaded operators, 583
search functions, 584
 use of, 78–82, 370
Structured programming, 11,
 13–19
 ATM example using, 13–14
 car maintenance example
 using, 16–17
 defined, 11
 object-oriented programming
 versus, 13–19
Subtraction (-) operator, 59
switch statements, 118–122, 145
Syntax, C++ language, 37–39

Tools for C++ development
 environment, 527
Top-down programming, *see*
 Structured programming
Trailer value, 568–569
Two-dimensional arrays, *see*
 Multidimensional arrays

Unary operators, 101–102,
 412–418
conditional statements, as,
 101–102
defined, 101
overloaded, 412–418
Undeclared identifiers, 198–200
unsigned data type modifiers,
 43, 45
using namespace std
 statement, 33

VValues, C++ programming,
 42–43
Variable scope, 204
Variables, 42–43, 46–48, 56–58,
 137–138, 204–211, 245–249,
 249–250, 269, 294–295.
See also Data variables

array, 294–295
automatic, 205
block scope, 205
containers, 42–43
data, 245–249, 249–250, 269
data types and, 42–43
declaration, 46–48
initializing, 56–58
defined, 42–43
float (floating point value),
 137–138
global, 205–209
identifiers, 47
int (integer value), 137–138
labels, 42–43
local, 205
precedence of operations,
 56–58
scope, 204
static, 209–211
troubleshooting control
 statements using, 137–138
values, 42–43
writing functions using, 204–211

Vector class *<vector>*, 142–144,
 370, 586
constructors, 586
functions, 586
objects and, 370
use of, 142–144

Vectors, 142, 303–304
arrays, comparison with,
 303–304
defined, 142

Virtual functions, 490–500,
 500–506
defined, 491
game programming, 494–496
inheritance and, 490–500
pay calculations program
 example, 500–506
polymorphism and, 490–500
pure, 492
two vehicles program
 example, 496–500
use of, 494–500
virtual shapes program
 example, 491–494

W

while loops, 122–123, 128–132
Whitespace characters, 35–36
write() function, 401–402
Writing classes, 371–399
 access specifiers, 375–377
 calling functions, 381–382
 class constructor functions, 377–378
 class declaration, 374
 class definition, 374
 class member, 374, 399–404
 cout statement, 382
 date class program examples, 371–391
 errors in, 424–427
 get function, 391–395
 header file, 378–379
 main function, 379
 overloaded class constructor function, 377

overloaded class member functions, 427
PI calculator example of, 395–399
pre-processor directive, 378
private access specifiers, 375
public access specifiers, 375
scope operator, 384–375
set function, 391–395
undeclared identifiers, 424–426
understanding secrets of, 383–391
unresolved external symbol, 426–427
volume calculator, program example of, 391–395
Writing functions, 179–243
 call-by value, 195–198
 compiler error, 214, 215–216
 errors encountered, overview of, 213–216

function body, 186, 192–194
function declarations, 186–187, 191–193
function definition, 186–187, 192–193
function prototype, 186–178, 191–192
header line, 192–194
header line, 192–194, 215–216
input parameters, 184, 214
link error, 214–215
overloaded functions, 200–202
requirements for, 186–200
return statement, 193–194
undeclared identifiers, 198–200
variable scope, 204
variables, using, 204–211
wxWidgets, defined, 8

Z

Zero indexed arrays, 296

<p>Write formatted output to the screen The value of "PI" is 3.14159.</p>	<p>Use stringstream class to create a formatted string. Need to #include <sstream> There are 12 eggs in a carton. The value of "PI" is 3.14159.</p>
<pre>double pi = 3.14159265; cout.setf(ios::fixed); cout.precision(5); cout << "\n The value of \"PI\" is " << pi << "." << endl;</pre>	<pre>double pi = 3.14159265; int dozen = 12; stringstream ss; ss.setf(ios::fixed); ss.precision(5); ss << "\n There are " << dozen << " eggs in a carton."; ss << "\n The value of \"PI\" is " << pi << "." << endl; string result = ss.str();</pre>

<p>Make a vector of strings Need to #include <vector> and #include <string></p> <p>Fill a vector with names and show them.</p>	<p>Make a queue of integers Need to #include <queue></p> <p>Fill a queue with 10 random numbers between 0 and 99. Show the first in line, then pop it off the queue.</p>
<pre>vector<string> Names; Names.push_back("Bob"); Names.push_back("Mike"); Names.push_back("John"); for(int i = 0; i < Names.size(); ++i) { cout << Names.at(i) << endl; }</pre>	<pre>queue<int> rNums; int n = 0, rn; for(n = 0; n < 10; ++n) { rn = rand()%100; rNums.push(rn); } cout << "\n The queue size is " << rNums.size(); while(rNums.size() > 0) { cout << "\n Next is " << rNums.front(); rNums.pop(); }</pre>

Function.h	Functions.cpp	Driver.cpp
<pre>#ifndef _FUNC_H #define _FUNC_H void WriteHello(); int AskAge(); void Write(int age); #endif</pre>	<pre>#include <iostream> using namespace std; void WriteHello() { cout << "\n Hello"; } int AskAge() { int age; cout << "Age? "; cin >> age; return age; } void Write(int age) { cout << "\n You are" << age << "years old."; }</pre>	<pre>#include "Functions.h" int main() { int age; WriteHello(); age = AskAge(); Write(age); return 0; }</pre>

Date.h	Date.cpp	DateDriver.cpp
<pre>#ifndef _DATE_H #define _DATE_H class Date { private: int mon,day,yr; public: Date(); Date(int m, int d, int y; void Write(); }; #endif</pre>	<pre>#include "Date.h" #include <iostream> using namespace std; Date::Date() { mon = day = 1; yr = 2007; } Date::Date(int m, int d, int y) { mon = m; day = d; yr = y; } void Date::Write() { cout << "\n" << mon << "/" << day << "/" << yr << "\n"; }</pre>	<pre>#include "Date.h" int main() { Date d1; Date d2(2,5,2008); d1.Write(); d2.Write(); return 0; }</pre>